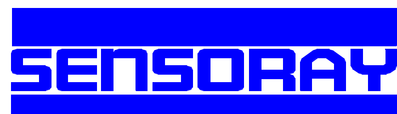
Model 626 Driver for Windows

September 12, 2002



Sensoray Co., Inc.
7313 SW Tech Center Dr., Tigard, Oregon 97223
voice: 503.684.8005, fax: 503.684.8164, e-mail: sales@sensoray.com
www.sensoray.com

Table of Contents

Introduction

Scope 1

Description 1

Block Diagram 1

Installation

Executable Software Components 2

Installation Procedure 2

Windows 98/2000/ME/XP 2

Windows NT4 3

SDK Components 3

Application SDK Components 3

Driver SDK Components 3

Fundamentals of Usage

Board Addressing 5

Board Handles 5

Physical Addressing 5

Thread-Safety 5

Channel Numbering 6

Programming Examples 6

Data Types 6

Required Function Calls 7

DLL Linking/Unlinking 7

Board Initialization 7

DLL Functions

Registration and Status Functions 8

S626_OpenBoard() 8

S626_CloseBoard() 10

S626_GetErrors() 11

S626_GetAddress() 12

Register Access Functions 12

S626_RegRead() 13

S626_RegWrite() 13

Analog Input Functions 14

S626_ResetADC() 14

S626_StartADC() 15

S626_WaitDoneADC() 15

S626_ReadADC() 15

A/D Programming Examples 16

Example: Scattered Acquisition 16

Example: Oversampling 16

Analog Output Functions 17

S626_WriteDAC() 17

Digital I/O (DIO) Functions 18

DIO Channel Groups 18

DIO Signal Polarity 18

S626_DIOModeSet() 18

S626_DIOModeGet() 19

S626_DIOWriteBankSet() 20

S626_DIOWriteBankGet() 20

S626_DIOReadBank() 21

S626_DIOEdgeSet() 22

S626_DIOEdgeGet() 22

S626_DIOCapEnableSet() 23

S626_DIOCapEnableGet() 24

S626_DIOCapStatus() 24

S626_DIOCapReset() 25

S626_DIOIntEnableSet() 26

S626_DIOIntEnableGet() 26

Table of Contents

Counter Functions 27

Counter Operating Modes 27

Input Signal Names 27

Using DIOs as Counter Outputs 27

Mode Descriptions 27

Configuration Options 28

IntSrc 28

LatchSrc 29

LoadSrc 29

IndxSrc 30

IndxPol 30

ClkSrc 30

ClkPol 31

ClkMult 31

ClkEnab 31

S626_CounterModeSet() 32

S626_CounterModeGet() 32

S626_CounterEnableSet() 33

S626_CounterPreload() 33

S626_CounterLoadTrigSet() 34

S626_CounterLatchSourceSet() 35

S626_CounterReadLatch() 35

S626_CounterCapStatus() 36

S626_CounterCapFlagsReset() 36

S626_CounterSoftIndex() 37

S626_CounterIntSourceSet() 37

Counter Programming Examples 38

Constants Used in Examples 38

Example: Periodic Interrupt Generator 39

Example: Encoder Interface 40

Example: Simple Event Counter 40

Example: Pulse Width Measurement 41

Example: Frequency Counter 42

Watchdog Timer Functions 44

S626_WatchdogPeriodSet() 44

S626_WatchdogPeriodGet() 44

S626_WatchdogEnableSet() 45

S626_WatchdogEnableGet() 45

S626_WatchdogReset() 46

S626_WatchdogTimeout() 46

Battery Functions 47

S626_BackupEnableSet() 47

S626_BackupEnableGet() 47

S626_ChargeEnableSet() 48

S626_ChargeEnableGet() 48

Interrupt Functions 49

S626_InterruptEnable() 49

S626_InterruptStatus() 50

Interrupt Programming Examples 51

Example: DIO Interrupts 51

Example: Counter Interrupts 52

Chapter 1: Introduction

1.1 Scope

This document discusses the contents and use of the distribution media supplied with Sensoray Model 626 boards. Primary focus is given to the installation and use of the API provided by the Windows driver for Model 626 boards, which is included on the distribution media. The driver works with the following Windows versions: 98, NT4, ME, 2000 and XP.

1.2 Description

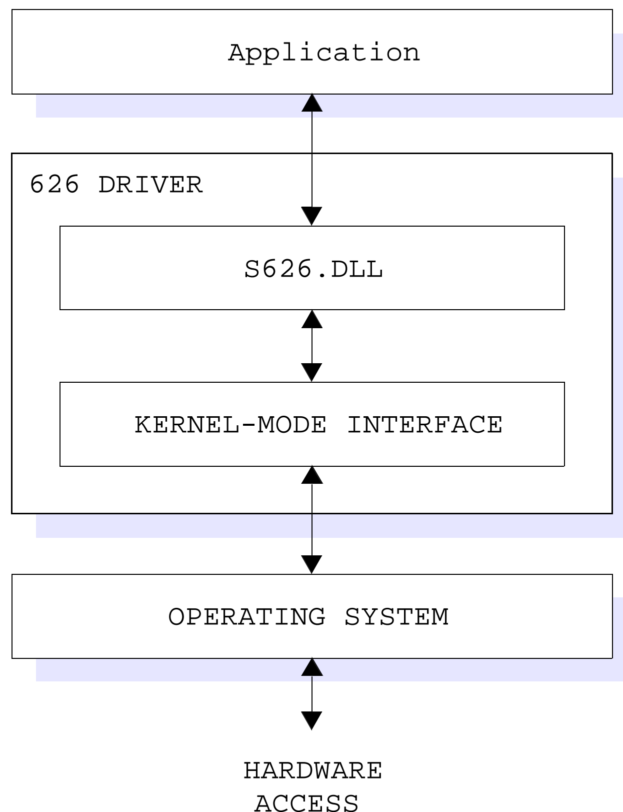
The Model 626 driver is a set of executable software modules that will interface one or more Sensoray Model 626 boards to a Windows application program of your design. Application programs may be developed in any popular Windows development environment, including Visual C++, Visual Basic, Delphi, etc.

Up to sixteen Sensoray Model 626 boards may be concurrently supported by the driver.

1.2.1 Block Diagram

The Model 626 driver consists of multiple software components that serve as an interface between the application program and the operating system. Figure 1 illustrates the hierarchical functional relationship between these software components.

Figure 1: Block diagram of the software hierarchy.



Chapter 2: Installation

2.1 Executable Software Components

The driver's executable software components must be correctly installed on the target system to ensure proper functioning of the Model 626 driver.

2.1.1 Installation Procedure

2.1.1.1 Windows 98/2000/ME/XP

The components that are required, and their locations in your system, are shown in Table 1.

Table 1: Required driver components for Windows 98/2000/ME/XP.

Filename	Directory	Function
`S626.DLL`	`SYSTEM`	Model 626 API
`SXDRV98.SYS`	`SYSTEM32\DRIVERS`	Kernel-mode driver

1. **Remove the old driver.** Perform the following steps *only* if you are installing the driver onto a computer that has an older version of the driver. If you do not have an older version of the 626 driver, proceed to step 2.
   a. Delete all instances of `WINDRVR.SYS` and `S626.DLL` from your system.
   b. Delete the old INF file for the model 626. For Windows 98, the file will be in <WINROOT>\INF\OTHER, with a filename containing "626." For Windows 2000, the file will be in <WINROOT>\INF, with the filename OEMx.INF (where x is a number); you must examine each such INF file with a text editor to determine which one applies to the model 626.
   c. Run the registry editor: use `RegEdt32.exe` if you are running Windows 2000, or `RegEdit.exe` if you are running Windows 98. If you are running Windows 2000 and Windows prevents you from performing one of the following registry changes, go to the Security menu and obtain authority to make the required registry key change.
   d. Delete the 626 board's registry key. This key has the following registry path:
   (Windows 2000) `HKEY_LOCAL_MACHINE|SYSTEM|CurrentControlSet|Enum|PCI`.
   (Windows 98) `HKEY_LOCAL_MACHINE/Enum/PCI`.
   The key that is to be deleted begins as follows: `VEN_1131&DEV_7146` ...
   e. Delete the old kernel-mode driver key. This key has the following registry path:
   (Windows 2000) `HKEY_LOCAL_MACHINE|SYSTEM|CurrentControlSet|Services|WinDrvr`.
   (Windows 98) `HKEY_LOCAL_MACHINE|SYSTEM|CurrentControlSet|Services|Class|WinDrvr`.
2. **Install the driver.** Perform the following steps:
   a. Locate all required software components in their appropriate directories as detailed in Section 2.1. Make sure there is only one instance of `SXDRV98.SYS` in your file system; there should be no duplicates of this file in your file system.
   b. Shut down the computer.
3. **Install the hardware.** Perform the following steps:
   a. With system power turned off, install the model 626 board(s) into the backplane.
   b. Apply system power and reboot.
   c. The New Hardware Detected dialog box will display. Instruct the system to "search for the best driver," then specify the path to the `SX.INF` file that is included on the distribution media. The New Hardware Wizard should find the file you specified and display its path in the dialog box. Instruct the Wizard to finish the installation.
   d. Reboot the system.

2.1.1.2 Windows NT4

The components that are required, and their locations in your system, are shown in Table 2.

Table 2: Required driver components for Windows NT4.

Filename	Directory	Function
S626.DLL	SYSTEM	Model 626 API
SXDRVNT.SYS	SYSTEM32\DRIVERS	Kernel-mode driver

1. **Install the driver.** Perform the following steps:
   a. Run the SETUPNT.BAT batch file. This will locate SXDRVNT.SYS in its required location (as shown above) and modify the Windows registry accordingly.
   b. Copy S626.DLL to the required target directory.
   c. Shut down the computer.
2. **Install the hardware.** Perform the following steps:
   a. With system power turned off, install the model 626 board(s) into the backplane.
   b. Apply system power and reboot.

2.2 SDK Components

2.2.1 Application SDK Components

The Model 626 distribution media includes several source-code modules that are designed to help you to accelerate the development of application programs that employ Model 626 boards:

WIN626.C Functions used for dynamically linking to and unlinking from S626.DLL. Compile this module and link the resulting object file to all Windows-based C/C++ applications that access functions in S626.DLL.

WIN626.H Declarations for resources supplied by WIN626.C. Include this file in all Windows-based C/C++ application modules that access S626.DLL functions.

APP626.H Essential data types and manifest constants required by 626-based applications. Include this file in all C/C++ application modules that access 626 boards.

WIN626.BAS Data types, constants, and function declarations required for Visual Basic applications. Include this file in any Visual Basic project that accesses functions in S626.DLL.

2.2.2 Driver SDK Components

Besides providing modules for Windows-based application program development, the distribution media also includes several platform-independent source-code modules that target custom driver development. Although the usage of these modules is beyond the scope of this document, they are briefly discussed here for the benefit of driver developers. Note: Sensoray does not provide technical support for custom driver developers who are using these source code modules.

CLS626.CPP Abstract C++ class, named Cls626, that serves as a foundation for developing a custom Model 626 driver for any platform.

CLS626.H Header file for the Cls626 class. Include this file in all C/C++ modules that reference the abstract class implemented in CLS626.CPP.

The `Cls626` class, declared in `CLS626.H` and implemented in `CLS626.C`, is intended to be the base class in a simple inheritance hierarchy. `Cls626` implements all board-specific, OS-independent functions for any driver. `Cls626` encapsulates all knowledge of Model 626 hardware and the machinations required to manipulate that hardware, but makes no assumptions about the OS platform upon which the board is operating.

At a minimum, two software components must be designed and implemented by the driver developer:

- A derived, OS-dependent interface class. This derived class, which is based upon `Cls626`, must supply `Cls626` with all OS-dependent hardware access and system interface functions that are declared as pure virtuals in the `Cls626` base class. This interface class implements such functions as dynamic memory allocation, kernel-mode interrupt processing and PCI BIOS interfacing.
- A driver Application Program Interface (API). This API, which is the software link between the application program and the driver, must create objects of the derived class type and expose the underlying public `Cls626` base class and derived class functions to application programs.

Chapter 3: Fundamentals of Usage

3.1 Board Addressing

3.1.1 Board Handles

Each board is assigned a board number, which is referenced in this document as a "handle." A handle is the logical address of a board. Most driver functions include the board handle as a parameter so that driver calls will be directed to a specific board. Since the driver supports up to sixteen Model 626 boards, valid handles may have any numerical value in the range 0 to 15.

Board handles are not OS-allocated handles in the traditional Windows sense, but rather are integer values that are assigned by the application program. When a board is first declared to the driver by the application program, any valid, unused handle may be specified for that board. Once a handle has been assigned to a board, it must not be used by any other board.

3.1.2 Physical Addressing

In addition to the board handle, which is the "logical" address for a board, each board also has a physical address. A board's physical address is always specified as a 32-bit, unsigned integer value. The physical address is a composite value that indicates both the PCI slot number and PCI bus number that the board resides in. The high word (16 bits) of the address value represents the bus number and the low word represents the slot number within that bus. For example, the physical address value `0x0002000A` indicates that the board is located in bus number 2, slot 10.

It is not always necessary to know the physical address of a board. A system containing a single Model 626 board, for example, has no need to know the location of the board; it is truly "plug and play." If two or more Model 626 boards are present in a system, however, it is essential that some means be provided to distinguish between the boards; that means is the physical address.

Note: PCI slot and bus numbers referenced by `S626.DLL` are generated by the PCI BIOS and consequently may differ from the assigned ordinal slot and bus numbers.

Two of the driver functions make use of physical board addresses:

- `S626_OpenBoard()` declares a board to the driver in order to establish a connection to the board.
- `S626_GetAddress()` returns the physical address of a connected board.

3.2 Thread-Safety

All driver functions are inherently thread-safe for multi-threaded, single-process applications. Although the driver is thread-safe, it does not allow multiple concurrent access to driver functions on a single board. The driver enforces this restriction. In other words, in multithreaded applications, although any thread can call any driver function at any time, the driver may block another thread from accessing the same board until the original thread leaves the driver.

The driver does not block if two threads concurrently access two different boards. However, if two threads concurrently access a single board, there is potential for blocking.

To guarantee thread-safe behavior in multi-process applications, two processes must never share access, concurrent or otherwise, to a single Model 626 board. The application is responsible for enforcing this restriction.

3.3 Channel Numbering

Many of the I/O resource classes provided by the Model 626 board have multiple instances of I/O circuitry which are referred to as *channels*. For example, the board has four analog output channels and 48 digital I/O channels.

Each DLL function that accesses a specific channel requires a channel number argument to designate the channel to be affected by the function. By convention, channel numbers always begin at zero and extend upward to the maximum valid channel number belonging to the addressed board:

- Digital I/O channel numbers range from 0 to 47.
- Analog Input channel numbers range from 0 to 15.
- Analog Output channel numbers range from 0 to 3.
- Counter channel numbers range from 0 to 5, as shown in Table 3.

Table 3: Counter channel numbering assignments.

Channel Number	0	1	2	3	4	5
Addressed Counter	0A	1A	2A	0B	1B	2B

3.4 Programming Examples

The C++ programming language has been used to code all programming examples. In most cases, programming examples can be easily adapted to other programming languages.

3.4.1 Data Types

All data values that are passed to or returned from DLL functions belong to a small set of data types. The data types employed by the API are listed in Table 4. Data types are referenced by their type names as shown in the left column of the table.

Table 4: Data types used by DLL functions

Type Name	Description
BYTE	8-bit, unsigned integer
WORD	16-bit, unsigned integer
SHORT	16-bit, signed integer
COUNTER_SETUP	16-bit, unsigned integer. See Section 4.6.2 for details.
DWORD	32-bit, unsigned integer
HBD	32-bit, unsigned integer
LONG	32-bit, signed integer
DOUBLE	8-byte, double-precision floating point value.
FPTR_ISR	Address of a function that takes no arguments and returns no value.

3.5 Required Function Calls

3.5.1 DLL Linking/Unlinking

Any application that accesses functions in `S626.DLL` must dynamically link to the DLL before calls are made to any of the DLL functions. Such applications must also unlink from the DLL when the application is terminated so that resources used by the DLL will be released. The means by which DLL linking and unlinking is implemented depends on your development environment.

If you are developing an application using C/C++, your application must call `S626_DLLOpen()` to dynamically link to the DLL before calling any DLL functions, and `S626_DLLClose()` to unlink from the DLL when the application terminates. These two functions are provided for C/C++ developers in the `WIN626.C` module on the distribution media. If you are using a language other that C/C++ that does not perform automatic DLL linking, you must provide functions equivalent to `S626_DLLOpen()` and `S626_DLLClose()` in your source language format.

Applications developed in Visual Basic do not require calls to equivalent `S626_DLLOpen()` or `S626_DLLClose()` functions because VB automatically links to a DLL when any DLL function is first called, and automatically unlinks from a previously linked DLL when the application terminates.

3.5.2 Board Initialization

Some DLL functions are used universally in all applications, while others, depending on application requirements, may or may not be used. All application programs must, as a minimum, perform the following steps for each Model 626 board:

1. Call `S626_OpenBoard()` to enable communication with the Model 626 board.
2. Call `S626_GetErrors()` to verify that the board is properly initialized, fault-free and ready to communicate.

Chapter 4: DLL Functions

4.1 Registration and Status Functions

4.1.1 S626_OpenBoard()

Function: Registers a Model 626 board with the driver.

Prototype: `VOID S626_OpenBoard( HBD hbd, DWORD address, FPTR callback, DWORD priority );`

Parameter	Type	Description
`hbd`	`HBD`	Board handle. Use any value between 0 and 15, inclusive, but do not use a value that has already been used for another Model 626 board.
`address`	`DWORD`	Composite physical address (as described in section 3.1.2) of the Model 626 board. The high 16-bit word contains the PCI bus number and the low word contains the PCI slot number. Set to zero to force the driver to detect and register any Model 626 board. You may then determine the address of the board by calling `S626_GetAddress()`.
`callback`	`FPTR_ISR`	Address of the application's interrupt callback function (see Section 4.10.1 for a discussion of interrupt callback functions). Set to zero if interrupts will not be used.
`priority`	`DWORD`	Priority level of the interrupt handler thread. See below for details.

The `priority` argument may be set to one of the following values:

Value	Priority level
`-2`	`THREAD_PRIORITY_LOWEST`
`-1`	`THREAD_PRIORITY_BELOW_NORMAL`
`0`	`THREAD_PRIORITY_NORMAL`. Specify this value if interrupts will not be used.
`1`	`THREAD_PRIORITY_ABOVE_NORMAL`
`2`	`THREAD_PRIORITY_HIGHEST`
`3`	`THREAD_PRIORITY_TIME_CRITICAL`

Returns: None.

Notes: This function registers a Model 626 board so that communication between the application program and board hardware will be enabled. Each board must be registered by this function before calling any other DLL functions that reference that board.

After registering a board, you may re-register the board by calling `S626_OpenBoard()` with the same board handle. This has the effect of calling `S626_CloseBoard()` to unregister the board (see Section 4.1.2), and then calling `S626_OpenBoard()` to register and reset the board. This feature can be used to automate recovery from soft failures that may have set "unrecoverable" error flags on a board.

Do not register two different boards (with two different handles) at the same physical address. This will result in unpredictable behavior and may cause your system to become unstable.

When you specify an address value equal to zero, the driver will seek and register the first Model 626 board in your system. This feature is useful in either of two cases:

1. You have only one Model 626 board in your system, therefore the application does not need to know the

board's physical location in the system. In this case, your application can simply specify zero for the `address` value.

2. You have two or more Model 626 boards in your system and you are unsure of the physical address values to specify in your application program. In this case, you may insert a Model 626 board into one of the PCI slots in which it will be residing when you execute your application, then call `S626_OpenBoard()` with `address` set to zero, followed by a call to `S626_GetAddress()` to determine the physical location of the board. The single Model 626 board is then moved to the slot in which the next Model 626 board will reside, and this process is repeated to learn the next physical address, etc. Once all of the physical addresses are known, they are "hardwired" into your application program and all of the target slots may be populated.

`S626_OpenBoard()` configures all board I/O resources as if a PCI bus hardware reset had occurred. The bullet items below detail the board configuration resulting from execution of this function. The last two items, battery backup and counter subsystems, are left unmodified in order to support battery backup of the counters during system power failure.

- The board's master interrupt is masked (disabled).
- The watchdog timer is disabled and the timer interval is set to 0.125 seconds.
- All digital I/O (DIO) outputs are programmed to the inactive state.
- Edge capturing is disabled on all DIO channels.
- Interrupts are disabled on all DIO channels.
- All DIO event capture edge polarities are set to zero (rising edges selected).
- DIO channel 0-5 are configured to operate as standard DIOs (vs. counter overflow outputs).
- All analog output channels are programmed to zero volts out.
- No ADC poll list is in effect; `S626_ResetADC()` must be called before `S626_ReadADC()`.
- Battery charging is disabled.
- Battery backup is unmodified.
- Counter cores, status and control registers are unmodified, except that all counter interrupts are disabled.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Declare board number 0, address unknown, no interrupts.
/////////////////////////////////////////////////////////////////////////////////

S626_OpenBoard( 0, 0, 0, 0 );

if ( S626_GetErrors( 0 ) )
{
    // .... Handle error ....
}
```

Example:

```
////////////////////////////////////////////////////////////////////////////////
// Declare board number 2, which resides at bus 0, slot 11, no interrupts.
////////////////////////////////////////////////////////////////////////////////

#define  BUS     0
#define  SLOT   11

S626_OpenBoard( 2, ( BUS << 16 ) | SLOT, 0, 0 );

if ( S626_GetErrors( 2 ) )
{
    // .... Handle error ....
}
```

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Declare board number 3, which resides at bus 1, slot 10, with interrupts.
// IntFunc() is the application ISR callback function.
/////////////////////////////////////////////////////////////////////////////////

S626_OpenBoard( 3, 0x0001000A, IntFunc, THREAD_PRIORITY_NORMAL );

if ( S626_GetErrors( 3 ) )
{
    // .... Handle error ....
}
```

4.1.2 S626_CloseBoard()

Function: Unregisters a Model 626 board with the driver.

Prototype: `VOID S626_CloseBoard( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board handle of the board to be unregistered.

Returns: None.

Notes: This function unregisters a Model 626 board that has been previously registered by `S626_OpenBoard()`. When called, this function will sever the driver's communication link between the application program and board hardware, and the board handle will be freed. Once freed, the board handle is available for assignment to the same board or to any other Model 626 board.

Each Model 626 board that has been registered by `S626_OpenBoard()` must be unregistered when its hardware resources are no longer needed by an application. This can be accomplished by calling `S626_CloseBoard()`.

`S626_CloseBoard()` modifies I/O resource states as follows:

- The board's master interrupt is masked (disabled).
- The watchdog timer is disabled.
- Battery charging is disabled.

Except as listed above, `S626_CloseBoard()` does not modify the states of board resources. If any board resources (i.e., analog output voltages or digital outputs) are to be programmed to application-defined "shutdown" states, the application must call the appropriate driver functions to program these states before calling `S626_CloseBoard()`.

Example:

```
/////////////////////////////////////////////////////////////////////////////
// Unregister board number 0 after programming analog and digital outputs
// to their application-defined "shutdown" states.
/////////////////////////////////////////////////////////////////////////////

// Program all analog outputs to zero volts.
for ( WORD DacChan = 0; DacChan < 4; S626_WriteDAC( 0, DacChan++, 0 );

// Program all digital outputs to the inactive state.
for ( WORD Group = 0; Group < 3; S626_DIOWriteBankSet( 0, Group++, 0 );

// Unregister the board and release its handle.
S626_CloseBoard( 0 );
```

4.1.3 S626_GetErrors()

Function: Returns error flags and clears to zero all resettable error flags.

Prototype: `DWORD S626_GetErrors( HBD hbd );`

Parameter	Type	Description
`hbd`	`HBD`	Board handle.

Returns: DWORD consisting of zero or more active-high bit flags. Each bit flag is an indicator for a specific error condition as designated by the following bit masks:

Bit Mask	Symbolic Name	Description
`0x00000001`	ERR_OPEN	Failed to open the kernel-mode driver. This is usually caused by a missing or unregistered software component. Make sure all required driver software components are properly installed and registered and no unnecessary driver components are installed or registered. See Chapter 2 for details.
`0x00000002`	ERR_CARDREG	Kernel-mode driver can't register the board. This is usually caused by applications that fail to unlink from `S626.DLL` when terminating. If this is not the case, the kernel-mode driver may be improperly installed or registered. See Chapter 2 for details.
`0x00000004`	ERR_ALLOC_MEMORY	Memory allocation error. This is typically caused by insufficient system RAM.
`0x00000008`	ERR_LOCK_BUFFER	DMA buffer lock failed.
`0x00000010`	ERR_THREAD	System failed to launch the interrupt thread.
`0x00000020`	ERR_INTERRUPT	Kernel-mode driver failed to enable interrupt.
`0x00000040`	ERR_LOST_IRQ	Missed interrupt. This is the result of interrupts that are occuring at too high a rate for the CPU to handle.
`0x00000080`	ERR_INIT	Failed to instantiate board object. This is typically caused by insufficient system RAM.
`0x00000100`	ERR_VERSION	Incompatible version of kernel-mode driver. Upgrade to the latest version of SXDRVR to resolve this problem.
`0x00010000`	ERR_ILLEGAL_PARM	Illegal function argument value (i.e., invalid DAC or ADC channel number was specified).
`0x00020000`	ERR_I2C	Board EEPROM access fault during board initialization. If this occurs, try unregistering the board and then re-registering it. If the error persists, the problem is likely a hardware fault that requires board repair.
`0x00100000`	ERR_DACTIMEOUT	DAC communication time-out.
`0x00200000`	ERR_COUNTERSETUP	Counter parameter is illegal in current counter operating mode.
`0x00400000`	ERR_DEBI_TIMEOUT	Local bus access fault.

Notes: This function may be called at any time after the target board has been opened with `S626_OpenBoard()`.

The symbolic names in the above error flag list are the names used to reference the bit mask values in the supplied C language source code. The definitions for these symbolic names may be found in the `APP626.H` source file.

Error flags with mask values of `0x00010000` and higher are associated with *recoverable* errors and are automatically reset to zero when `S626_GetErrors()` executes. Error flags with mask values below

`0x00010000` are associated with unrecoverable errors and can be cleared only by unregistering the board and then re-registering it. See Section 4.1.1 for details.

Once an error flag is set, it remains set until cleared by re-registering the board, or, in the case of recoverable errors, `S626_GetErrors()` is called.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Fetch all error flags from board number 2 and reset recoverable error flags.
///////////////////////////////////////////////////////////////////////////////

DWORD faults = S626_GetErrors( 2 );
```

4.1.4 S626_GetAddress()

Function: Returns the physical address of a Model 626 board.

Prototype: `DWORD S626_GetAddress( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board handle.

Returns: DWORD value consisting of the PCI bus number in the high 16-bit word and PCI slot number in the low word.

Notes: This function serves primarily as a utility to help application developers determine the slot and bus numbering conventions of a PCI system. In PCI systems that have multiple Model 626 boards, the ability to reference a specific board is mandatory. For example, one board might control a heater while a second board controls a cooler. For obvious reasons, the application must be able to distinguish between these boards, and the only way to do this is to assign fixed physical addresses to each board.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Determine the physical addresses of a Model 626 board.  With no other
// Model 626 boards installed in the system, a single Model 626 board
// is inserted into the target PCI slot, then this code is executed to
// determine the slot's physical address.
///////////////////////////////////////////////////////////////////////////////

// Declare board to driver; set address to zero to force the driver to locate
// a Model 626 board.  Interrupts will not be used on this board.
S626_OpenBoard( 0, 0, 0, 0 );

// Handle any errors.
if ( S626_GetErrors( 0 ) )
{
    // .... Handle error ....
}

// Fetch composite address from driver.
DWORD adrs = S626_GetAddress( 0 );

// Display PCI bus and slot numbers.
printf( "Bus no. = %d, Slot no. = %d\n", adrs >> 16, adrs & 0xFFFF );
```

4.2 Register Access Functions

The register access functions may be used to directly read from or write to any register in the Model 626 board's local address space. These functions serve primarily as diagnostics since nearly all of the board's functional features can be manipulated from other, higher-level functions provided by the DLL.

4.2.1 S626_RegRead()

Function: Returns a value from a Model 626 local bus hardware register.

Prototype: `WORD S626_RegRead( HBD hbd, WORD regadrs );`

Parameter	Type	Description
hbd	HBD	Board handle.
regadrs	WORD	Address of the target hardware register expressed as an offset from the base address of the board's local register space.

Returns: WORD value consisting of the 16-bit value read from the target hardware register.

Notes: Use this function to read the contents of a local bus hardware register on a Model 626 board.

Example:

```
///////////////////////////////////////////////////////////////////////////
// Read the states of digital inputs channels 0-15 from board number 0.
///////////////////////////////////////////////////////////////////////////

#define REG_RDDINA    0x0040      // Port address used to read DIO group 0 inputs.

// Capture a snapshot of the input states of digital I/O channels 0-15.
WORD InputStates = S626_RegRead( 0, REG_RDDINA );
```

4.2.2 S626_RegWrite()

Function: Writes a value into a Model 626 local bus hardware register.

Prototype: `VOID S626_RegWrite( HBD hbd, WORD regadrs, WORD value );`

Parameter	Type	Description
hbd	HBD	Board handle.
regadrs	WORD	Address of the target hardware register expressed as an offset from the base address of the board's local register space.
value	WORD	Data value to be written to the hardware register.

Returns: None.

Notes: Use this function to write a value into a local bus hardware register on a Model 626 board.

Example:

```
///////////////////////////////////////////////////////////////////////////
// Write all zeros to the MISC1 register on board number 2.
///////////////////////////////////////////////////////////////////////////

#define REG_MISC1    0x0088      // Port address of MISC1 control register.

// Write zeros to the MISC1 register.
S626_RegWrite( 2, REG_MISC1, 0 );
```

4.3 Analog Input Functions

4.3.1 S626_ResetADC()

Function: Initializes the analog-to-digital converter in preparation for digitizing analog inputs.

Prototype: `VOID S626_ResetADC( HBD hbd, BYTE *pollist );`

Parameter	Type	Description
hbd	HBD	Board handle.
pollist	BYTE*	Address of a poll list.

Returns: None.

Notes: Before executing any A/D conversions, a "poll" list must be created. The poll list is a schedule of analog-to-digital acquisitions that will transpire when `S626_ReadADC()` is called.

The poll list contains at least one and as many as sixteen acquisition items. Each item has two components: the analog input channel number to be digitized and the gain to be applied during the measurement. The last item in the list is marked with an End Of Poll List (EOPL) flag. When an A/D conversion is started, all items in the poll list are digitized in the order in which they appear in the list. Channel ordering is purely arbitrary and is at the discretion of the application programmer.

After populating or modifying the poll list, `S626_ResetADC()` must be called to initialize the ADC and to pass to it the poll list. When this has been done, `S626_ReadADC()` may be called to invoke the actual A/D conversions. Calls to `S626_ReadADC()` may be repeated as desired so long as the poll list is not changed. If a new set of channels is required or any measurement ranges need to be changed, the poll list must be modified and `S626_ResetADC()` must be called again before the next call to `S626_ReadADC()`.

It is possible to digitize a channel more than once during a single scan by repeating the channel in the poll list. If an analog input channel must be converted repeatedly (i.e., for oversampling), it may be faster to duplicate the channel in the poll list; this strategy will return up to sixteen successive digitized values from the channel with a single call to `S626_ReadADC()`.

Each poll list item occupies one byte, as shown in Figure 2.

Figure 2: Organization of a byte in an A/D poll list.

- `EOPL` is set to 1 to mark the end of the poll list, or 0 to indicate more poll list items follow the current item.
- `GAIN` sets the ADC full-scale input range: `0` selects the `±10V` range, `1` selects the `±5V` range.
- `CHANNEL NUMBER` specifies the analog input channel in the range 0 through 15.

Example: See "A/D Programming Examples" on page 16.

4.3.2 S626_StartADC()

Function: Initiates digitization of all acquisition items in an ADC poll list.

Prototype: `VOID S626_StartADC( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board handle.

Returns: None.

Notes: This function initiates a sequence of A/D conversions, as specified by a poll list, and then immediately returns without waiting for the sequence to run to completion. The conversion process will then run autonomously while the application performs other tasks.

When called, this function launches an I/O channel program that runs on a dedicated I/O processor residing on the Model 626 board. The I/O processor runs independently of the system processor so that, regardless of what is happening in the Windows environment, the time interval between the A/D acquisitions specified in the poll list will be deterministic.

Before calling this function, `S626_ResetADC()` must be called to convert the target poll list into an equivalent I/O channel program. In addition, `S626_ResetADC()` must be called again if the poll list is modified in any way.

Example: See Section 4.3.5.

4.3.3 S626_WaitDoneADC()

Function: Retrieves the digitized values resulting from a call to `S626_StartADC()`.

Prototype: `VOID S626_WaitDoneADC( HBD hbd, SHORT *databuf );`

Parameter	Type	Description
hbd	HBD	Board handle.
databuf	SHORT*	Address of an array that will receive the digitized data.

Returns: None.

Notes: Before calling this function, `S626_StartADC()` must be called to invoke the conversion process. This function stores the resulting signed, digitized data values in the `databuf[]` array. The calling thread will be blocked until the conversion process has finished and digitized data has been transferred to `databuf[]`.

Example: See Section 4.3.5.

4.3.4 S626_ReadADC()

Function: Digitizes all acquisition items in an ADC poll list.

Prototype: `VOID S626_ReadADC( HBD hbd, SHORT *databuf );`

Parameter	Type	Description
hbd	HBD	Board handle.
databuf	SHORT*	Address of an array that will receive the digitized data.

Returns: None.

Notes: This function performs a sequence of A/D conversions, as specified by a poll list, and stores the signed, digitized data values in the `databuf[]` array. It is the equivalent of calling two other functions in sequence: `S626_StartADC()` and `S626_WaitDoneADC()`.

When called, this function launches an I/O channel program that runs on a dedicated I/O processor residing on the Model 626 board. The I/O processor runs independently of the system processor so that, regardless of what is happening in the Windows environment, the time interval between the A/D acquisitions specified in the poll list will be deterministic.

Before calling this function, `S626_ResetADC()` must be called to convert the target poll list into an equivalent I/O channel program. In addition, `S626_ResetADC()` must be called again if the poll list is modified in any way.

Example: See Section 4.3.5.

4.3.5 A/D Programming Examples

4.3.5.1 Example: Scattered Acquisition

```
///////////////////////////////////////////////////////////////////////////////
// Digitize channels 2, 3 and 6 on board number 0.
///////////////////////////////////////////////////////////////////////////////

#define RANGE_10V      0x00    // Range code for ADC ±10V range.
#define RANGE_5V       0x10    // Range code for ADC ±5V range.
#define EOPL           0x80    // ADC end-of-poll-list marker.
#define CHANMASK       0x0F    // ADC channel number mask.

// Allocate data structures.  We allocate enough space for maximum possible
// number of items (16) even though this example only has 3 items.  Although
// this is not required, it is the recommended practice to prevent programming
// errors if your application ever modifies the number of items in the poll list.
BYTE poll_list[16];                        // List of items to be digitized.
SHORT databuf[16];                         // Buffer to receive digitized data.

// Populate the poll list.
poll_list[0] = 2 | RANGE_10V;              // Chan 2, ±10V range.
poll_list[1] = 3 | RANGE_5V;               // Chan 3, ±5V range.
poll_list[2] = 6 | RANGE_10V | EOPL;       // Chan 6, ±10V range, mark as list end.

// Prepare for A/D conversions by passing the poll list to the driver.
S626_ResetADC( 0, poll_list );

// Digitize all items in the poll list.  As long as the poll list is not modified,
// S626_ReadADC() can be called repeatedly without calling S626_ResetADC() again.
// This could be implemented as two calls: S626_StartADC() and S626_WaitDoneADC().
S626_ReadADC( 0, databuf );

// Display all digitized binary data values.
for ( int i = 0; i <= 2; i++ )
    printf( "Channel %d = %d\n", poll_list[i] & CHANMASK, databuf[i] );
```

4.3.5.2 Example: Oversampling

```
///////////////////////////////////////////////////////////////////////////////
// Average sixteen measurements from analog input channel 2, board number 0.
// Channel 2 will be digitized on the ±10V input range.
///////////////////////////////////////////////////////////////////////////////

#define EOPL           0x80    // ADC end-of-poll-list marker.

SHORT databuf[16];      // Buffer to receive digitized data.
```

```
// Create and populate a poll list to digitize channel 2 sixteen times.
BYTE poll_list[] = { 2,2,2,2, 2,2,2,2, 2,2,2,2, 2,2,2,2|EOPL };

// Prepare for A/D conversions by passing the poll list to the driver.
S626_ResetADC( 0, poll_list );

// Digitize all items in the poll list.  As long as the poll list is not modified,
// S626_ReadADC() can be called repeatedly without calling S626_ResetADC() again.
S626_ReadADC( 0, databuf );

// Compute and display the average of the 16 measured values.
LONG sum   = 0;
WORD *pbuf = databuf;
for ( int i = 0; i < 16; sum += (LONG)( *pbuf++ ) );
printf( "Average A/D value = %d\n", (WORD)( sum >> 4 ) );
```

4.4 Analog Output Functions

4.4.1 S626_WriteDAC()

Function: Programs an analog output setpoint on one DAC channel.

Prototype: `VOID S626_WriteDAC( HBD hbd, WORD chan, LONG setpoint );`

Parameter	Type	Description
hbd	HBD	Board handle.
chan	WORD	Channel number of the DAC that will be programmed to the new setpoint. Valid channel numbers range from 0 through 3. Specifying a DAC channel number greater than 3 will cause the ERR_ILLEGAL_PARM error flag to be set
setpoint	LONG	New setpoint value. Values may range from -8191 to +8191.

Returns: None.

Notes: Setpoint values that exceed the valid value range will be limited to the legal range; values less than -8191 will be forced to -8191, and values greater than +8191 will be forced to +8191.

Example:

```
////////////////////////////////////////////////////////////////////////////
// This function sets a DAC channel to the specified output voltage.
////////////////////////////////////////////////////////////////////////////

#define DAC_VSCALAR    819.1      // Binary-to-volts scalar for DAC.

SetDacVoltage( HBD hbd, WORD channel, DOUBLE volts )
{
    // Make adjustments to prevent conversion errors.
    if ( volts > 10.0 )         volts =  10.0;
    else if ( volts < -10.0 )   volts = -10.0;

    // Program new DAC setpoint.
    S626_WriteDAC( hbd, channel, (LONG)( volts * DAC_VSCALAR ) );
}
```

4.5 Digital I/O (DIO) Functions

4.5.1 DIO Channel Groups

In many of the DIO functions, DIO channels are addressed and manipulated in sets of sixteen channels called *groups*. Model 626 has a total of 48 DIO channels, numbered 0 through 47, which are affiliated with three 16-channel DIO groups, designated as group numbers 0 through 2. Each channel is assigned a fixed bit position within a 16-bit integer value that encapsulates the group. The relationships between groups and channels are shown in Table 5.

Table 5: Relationships between DIO groups and channels.

DIO Group Number	DIO Channel Range	Bit Assignments of DIO Channels in Group Word															
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0 to 15	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	16 to 31	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
2	32 to 47	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

For example, consider DIO channel group number 2. Any access to this group will be in the form of a sixteen-bit word whose most significant bit corresponds to DIO channel 47 and least significant bit corresponds to DIO channel 32.

4.5.2 DIO Signal Polarity

Each DIO channel is electrically configured to function as an open-collector, active low, "wired-OR" circuit. This means that a DIO channel is actively driven to zero volts when it is turned "on," and is passively pulled up to +5 Volts when it is turned "off." In the context of the *logical* state of a DIO channel, however, logic zero represents the "off" state and logic one represents the "on" state.

Table 6: Equivalent representations of DIO signal states.

Generic	Physical	Logical
Active (on)	0V, active driver	1
Inactive (off)	+5V, passive driver	0

To avoid confusion, all references to DIO channel states are restricted to either the logical states, *1* and *0*, or the equivalent generic terms *active* and *inactive*.

4.5.3 S626_DIOModeSet()

Function: Sets the operating mode of a dual-mode DIO channel.

Prototype:
```
VOID S626_DIOModeSet( HBD hbd, WORD chan, WORD overflow );
```

Parameter	Type	Description
`hbd`	`HBD`	Board handle.
`chan`	`WORD`	DIO channel number, in the range 0 to 5, for which the operating mode is to be programmed. Specifying an illegal channel number will cause the `ERR_ILLEGAL_PARM` error flag to be set.
`overflow`	`WORD`	Indicates whether the specified DIO channel is to operate as a standard DIO (zero) or as a counter overflow output (non-zero).

Returns: None.

Notes: Each of DIO channels 0 through 5 can independently function as either a DIO or as a counter overflow output. Table 7 shows the relationship between DIO channel number and associated counter channel number. For example, DIO channel 4 can behave either as a standard DIO or as the overflow output for counter 2A (counter channel number 2).

Table 7: Counter channels associated with dual-function DIO channels.

DIO Channel :	**0**	**1**	**2**	**3**	**4**	**5**
Counter Channel :	0A	0B	1A	1B	2A	2B

When configured as a counter overflow output, a DIO will produce a single, 500 nanosecond wide output pulse when the associated counter overflows.

Example:

```
//////////////////////////////////////////////////////////////////////////////////
// Configure board 0, DIO channel 2 as overflow output for counter channel 1A.
//////////////////////////////////////////////////////////////////////////////////

#define DIO_STANDARD    0
#define DIO_OVERFLOW    1

S626_DIOModeSet( 0, 2, DIO_OVERFLOW );
```

4.5.4 S626_DIOModeGet()

Function: Returns the operating mode of a dual-mode DIO channel.

Prototype: `WORD 626_DIOModeGet( HBD hbd, WORD chan );`

Parameter	**Type**	**Description**
hbd	HBD	Board handle.
chan	WORD	DIO channel number, in the range 0 to 5, from which the operating mode shall be returned. Specifying an illegal channel number will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value indicating the operating mode of the specified DIO channel. *Zero* indicates the channel is operating as a standard DIO, *non-zero* indicates the channel is operating as a counter overflow output (see Table 7 for associated counter channel numbers).

Example:

```
//////////////////////////////////////////////////////////////////////////////////
// Display the operating mode of board number 0, DIO channel 2.
//////////////////////////////////////////////////////////////////////////////////

printf( S626_DIOModeGet( 0, 2 ) ? "Overflow Mode" : "Standard Mode" );
```

4.5.5 S626_DIOWriteBankSet()

Function: Modifies the output states of a DIO channel group.

Prototype: `VOID S626_DIOWriteBankSet( HBD hbd, WORD group, WORD states );`

Parameter	Type	Description
`hbd`	`HBD`	Board handle.
`group`	`WORD`	Group number of the sixteen DIO channels that will be programmed to new state values. Legal values are 0, 1 or 2, as described in Section 4.5.1.
`states`	`WORD`	New logical state values to be written to the sixteen DIO channels belonging to `group`. Set a bit value to *zero* to program the corresponding DIO channel to the inactive state, or to *one* to program the channel to the active state. See Section 4.5.1 for a description of the bit position of each DIO channel in `states`. See Section 4.5.2 for a discussion of logical vs. physical channel states.

Returns: None.

Notes: This function *simultaneously* modifies the output states of all sixteen channels in the target DIO channel group.

Any DIO channel which is to serve as a digital input must be programmed to the inactive state. This causes the channel to be passively driven to the inactive state, thereby allowing an external signal source to assert control over the physical DIO state.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// On board number 0, set DIO channels 0-11 inactive and channels 12-15 active.
/////////////////////////////////////////////////////////////////////////////////

S626_DIOWriteBankSet( 0, 0, 0xF000 );     // Channels 0-15 are in group 0.
```

4.5.6 S626_DIOWriteBankGet()

Function: Returns the output states of a DIO channel group.

Prototype: `WORD S626_DIOWriteBankGet( HBD hbd, WORD group );`

Parameter	Type	Description
`hbd`	`HBD`	Board handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose output states are to be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the logical output states of all sixteen channels in the target DIO channel group. Each bit of the returned word represents the logical output state of the corresponding DIO channel: *zero* indicates the inactive state, and *one* indicates the active state. See Section 4.5.1 for a description of the bit position of each DIO channel in the returned value.

Notes: The returned value reflects the states of the board's DIO channel output drivers, which in normal operation are the same as the values last written by `S626_DIOWriteBankSet()`. This is in contrast with the values returned by `S626_DIOReadBank()`, which represents the *input* states present on the DIO channels. The value returned from `S626_DIOReadBank()` can be affected by external signal sources driving the DIO channels, while the value returned from `S626_DIOWriteBankSet()` is influenced only by the Model 626 output drivers.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Display the output states of DIO channels 16-31 on board number 0.
/////////////////////////////////////////////////////////////////////////////////
```

```
WORD states = S626_DIOWriteBankGet( 0, 1 );    // Channels 16-31 are in group 1.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d driver state = %d\n", i + 16, ( states >> i ) & 1 );
```

Example:

```
///////////////////////////////////////////////////////////////////////////////
// This function sets the specified DIO channel (chan numbers range from 0
// to 47) to the active state without affecting any other DIO channels.
///////////////////////////////////////////////////////////////////////////////

VOID DIO_SetToActive( HBD hbd, WORD chan )
{
    WORD group = chan >> 4;              // Group number associated with chan.
    WORD mask  = 1 << ( chan & 15 );    // Bit mask for chan within its group.

    S626_DIOWriteBankSet( 0, group, S626_DIOWriteBankGet( 0, group ) | mask );
}
```

4.5.7 S626_DIOReadBank()

Function: Returns the logical input states of a DIO channel group.

Prototype: `WORD S626_DIOReadBank( HBD hbd, WORD group );`

Parameter	Type	Description
hbd	HBD	Board Handle.
group	WORD	Group number of the sixteen DIO channels whose input states will be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the logical input states of all sixteen channels in the target DIO group. Each bit of the returned word represents the logical input state of the corresponding DIO channel: *zero* indicates the inactive state, and *one* indicates the active state. See Section 4.5.1 for a description of the bit position of each DIO channel in the returned word value.

Notes: The returned value reflects the driven states of the board's DIO inputs. The value is thus influenced by both the board's DIO output drivers and any external signal sources that may be physically driving DIO channels.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Display the logical input states of DIO channels 32-47 on board number 0.
///////////////////////////////////////////////////////////////////////////////

WORD states = S626_DIOReadBank( 0, 2 );   // Channels 32-47 are in group 2.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d input state = %d\n", i + 32, ( states >> i ) & 1 );
```

4.5.8 S626_DIOEdgeSet()

Function: Programs the polarity of edges that will trigger DIO event captures.

Prototype: `VOID S626_DIOEdgeSet( HBD hbd, WORD group, WORD edges );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose edge polarities will be programmed. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.
`edges`	`WORD`	Edge polarities to be used by the sixteen DIO channels belonging to `group`. Set a bit value to *zero* to select the active-to-inactive (falling) edge, or to *one* to select the inactive-to-active (rising) edge. See Section 4.5.1 for a description of the bit position of each DIO channel in `edges`.

Returns: None.

Notes: This function should be called to select the polarity of edge events to be captured before enabling captures. After programming the edge polarities, capturing may then be enabled by means of `S626_DIOCapEnableSet()`.

Only DIO channels 0 through 39 have edge capture capability. Consequently, `edges` bits that reference DIO channels 40 through 47 will be ignored when addressing channel group number 2.

Example:

```
////////////////////////////////////////////////////////////////////////////////
// On board 0, configure DIO channels 0-7 to capture DIO rising edge events,
// and channels 8-15 to capture falling edge events.
////////////////////////////////////////////////////////////////////////////////

S626_DIOEdgeSet( 0, 0, 0x00FF );     // Channels 0-15 are in group 0.
```

4.5.9 S626_DIOEdgeGet()

Function: Returns the polarity of edges that will trigger DIO event captures.

Prototype: `WORD S626_DIOEdgeGet( HBD hbd, WORD group );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose edge polarities are to be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the event capture edge polarities in use by the sixteen DIO channels belonging to `group`. As described in Section 4.5.1, each bit represents the edge polarity of one DIO channel: *zero* indicates active-to-inactive (falling) edge, and *one* indicates inactive-to-active (rising) edge.

Notes: Only DIO channels 0 through 39 have edge capture capability. Consequently, you should ignore any bits that reference DIO channels 40 through 47 when addressing channel group number 2.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Display board 5, DIO channel 0-15 edge polarities.
///////////////////////////////////////////////////////////////////////////////

WORD edges = S626_DIOEdgeGet( 5, 0 );    // Channels 0-15 are in group 0.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d edge = %d\n", i, ( edges >> i ) & 1 );
```

4.5.10 S626_DIOCapEnableSet()

Function: Enables/disables capturing of DIO edge events.

Prototype: `VOID S626_DIOCapEnableSet( HBD hbd, WORD group, WORD chans, WORD enable );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels that are to be enabled/disabled for event capturing. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.
`chans`	`WORD`	Bit flags indicating which of the sixteen DIO channels belonging to `group` are to be affected. See Section 4.5.1 for a description of the bit position of each DIO channel in `chans`. Set a bit flag to *zero* to exclude the associated channel, or to *one* to cause the channel's event captures to be enabled or disabled as specified by `enable`.
`enable`	`WORD`	Indicates whether DIO channels specified by `chans` are to have event captures enabled or disabled. Set to *zero* to disable event captures, or to *one* to enable event captures.

Returns: None.

Notes: In order for edge events to be captured on a specific channel, the channel must have edge event capturing enabled by this function. Typically, the edge polarity is first selected by means of `S626_DIOEdgeSet()`, and then `S626_DIOCapEnableSet()` is called to enable captures of the selected edge.

Only DIO channels 0 through 39 have edge capture capability. Consequently, `chans` bits that reference DIO channels 40 through 47 will be ignored when addressing channel group number 2.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// On board 4, enable event capturing on channels 0-2 and leave event
// capture enables unmodified on channels 3-15.  The chans value will
// be binary 0000_0000_0000_0111 (= decimal 7) to modify only channels 0-2.
///////////////////////////////////////////////////////////////////////////////

#define DIOCAP_ENABLE    1
#define DIOCAP_DISABLE   0

S626_DIOCapEnableSet( 4, 0, 7, DIOCAP_ENABLE );    // Channels 0-15 are in group 0.
```

4.5.11 S626_DIOCapEnableGet()

Function: Returns DIO capture enables.

Prototype: `WORD S626_DIOCapEnableGet( HBD hbd, WORD group );`

Parameter	Type	Description
hbd	HBD	Board Handle.
group	WORD	Group number of the sixteen DIO channels whose capture enables are to be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the event capture enables of the sixteen DIO channels belonging to `group`. As described in Section 4.5.1, each bit represents the edge polarity of one DIO channel: *zero* indicates channel event capturing is disabled, and *one* indicates capturing is enabled.

Notes: Only DIO channels 0 through 39 have edge capture capability. Consequently, you should ignore any returned bits that reference DIO channels 40 through 47 when addressing channel group number 2.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Display the states of DIO channel 0-15 capture enables on board number 5.
///////////////////////////////////////////////////////////////////////////////

WORD enables = S626_DIOCapEnableGet( 5, 0 );   // Channels 0-15 are in group 0.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d cap enable = %d\n", i, ( enables >> i ) & 1 );
```

4.5.12 S626_DIOCapStatus()

Function: Returns captured DIO events.

Prototype: `WORD S626_DIOCapStatus( HBD hbd, WORD group );`

Parameter	Type	Description
hbd	HBD	Board Handle.
group	WORD	Group number of the sixteen DIO channels whose capture status are to be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the event capture status of the sixteen DIO channels belonging to `group`. As described in Section 4.5.1, each bit represents the edge polarity of one DIO channel: *zero* indicates no edge events have been captured, and *one* indicates an edge event has been captured.

Notes: After capturing an edge event on a DIO channel, the associated capture status flag will be active and, if enabled by `S626_DIOIntEnableSet()`, an interrupt request will be asserted.

To reset a channel's capture status flag (and negate any affiliated interrupt request), you must disable its capture enable flag by calling `S626_DIOCapEnableSet()`. You may then call `S626_DIOCapEnableSet()` again to re-enable capturing for subsequent edge events.

Only DIO channels 0 through 39 have edge capture capability. Consequently, you should ignore any returned bits that reference DIO channels 40 through 47 when addressing channel group number 2.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Display the capture states of DIO channel 0-15 on board number 2.
/////////////////////////////////////////////////////////////////////////////////

WORD caps = S626_DIOCapStatus( 2, 0 );    // Channels 0-15 are in group 0.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d captured = %d\n", i, ( caps >> i ) & 1 );
```

4.5.13 S626_DIOCapReset()

Function: Resets DIO event capture flags.

Prototype: `VOID S626_DIOCapReset( HBD hbd, WORD group, WORD chans );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose capture flags are to be affected. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.
`chans`	`WORD`	Bit flags indicating which of the sixteen DIO channels belonging to `group` are to be affected. See Section 4.5.1 for a description of the bit position of each DIO channel in `chans`. Set a bit flag to *one* to reset the channel's event capture flag, or to *zero* to leave the channel's event capture flag in its current state.

Returns: None.

Notes: After capturing an edge event on a DIO channel, the associated capture status flag will be active and, if enabled by `S626_DIOIntEnableSet()`, an interrupt request will be asserted. Call this function to reset one or more DIO capture flags after the associated events have been processed. In addition to resetting the capture flags, `S626_DIOCapReset()` also clears any corresponding interrupt service requests.

Only DIO channels 0 through 39 have edge capture capability. Consequently, `chans` bits that reference DIO channels 40 through 47 will be ignored when addressing channel group number 2.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Reset event capture flags for board 2, DIO channels 0 and 1.
/////////////////////////////////////////////////////////////////////////////////

S626_DIOCapReset( 2, 0, 0x0003 );    // Channels 0 and 1 are in group 0.
```

4.5.14 S626_DIOIntEnableSet()

Function: Enables/disables interrupt requests in response to captured DIO edges.

Prototype: `VOID S626_DIOIntEnableSet( HBD hbd, WORD group, WORD enables );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose interrupt enables are to be modified. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.
`enables`	`WORD`	Bit flags indicating, for each of the sixteen DIO channels belonging to `group`, whether the captured edge interrupt shall be enabled or disabled. See Section 4.5.1 for a description of the bit position of each DIO channel in `chans`. Set a bit value to *zero* to disable the associated channel interrupt, or to *one* to enable the interrupt.

Returns: None.

Notes: This function enables or disables interrupt request generation in response to captured edges on DIO channels. In a typical application program, edge captures are configured and enabled by calling `S626_DIOEdgeSet()` and `S626_DIOCapEnableSet()` before invoking `S626_DIOIntEnableSet()` to enable capture interrupts.

Only DIO channels 0 through 39 have edge capture capability. Consequently, `enables` bits that reference DIO channels 40 through 47 will be ignored when addressing channel group number 2.

Example: See “Example: DIO Interrupts” on page 51.

4.5.15 S626_DIOIntEnableGet()

Function: Returns DIO interrupt enables.

Prototype: `WORD S626_DIOIntEnableGet( HBD hbd, WORD group );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`group`	`WORD`	Group number of the sixteen DIO channels whose interrupt enables will be returned. Legal values are 0, 1 or 2, as described in Section 4.5.1. Specifying a DIO group number greater than 2 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value containing the interrupt enables of the sixteen DIO channels belonging to `group`. As described in Section 4.5.1, each bit represents the interrupt enable of one DIO channel: *zero* indicates the interrupt is disabled, and *one* indicates the interrupt is enabled.

Notes: Only DIO channels 0 through 39 have the ability to generate interrupt requests in response to captured edge events. When addressing channel group number 2, all returned bits that reference DIO channels 40 through 47 should be ignored.

Example:

```
//////////////////////////////////////////////////////////////////////////
// Display the interrupt enables of DIO channel 0-15 on board number 2.
//////////////////////////////////////////////////////////////////////////

WORD caps = S626_DIOIntEnableGet( 2, 0 );    // Channels 0-15 are in group 0.

for ( int i = 15; i >= 0; i-- )
    printf( "DIO %d interrupt enable = %d\n", i, ( caps >> i ) & 1 );
```

4.6 Counter Functions

4.6.1 Counter Operating Modes

Counters may be programmed to operate in various behavioral configurations called *modes*. The following modes are supported by the Model 626 board:

- **Timer.** When operating in this mode, a counter will count either up or down at a constant rate.
- **Counter.** This mode supports external, two-phase or single-phase clock sources.

  In the case of a two-phase, quadrature-encoded clock source, count and direction controls are derived from the two clock phases. Clock multipliers of 1x, 2x and 4x are supported.

  In the case of a single-phase clock source, the counter behaves as a simple event counter. Count control is derived from the clock, and direction control is derived from a direction control input signal. Clock multipliers of 1x, 2x and 4x are legal, but use of the 2x and 4x multipliers is not recommended.
- **Extender.** This mode, which applies only to B counters, causes a counter to behave as an extension of a paired A counter that is configured as a Timer or Event Counter.

4.6.1.1 Input Signal Names

Each counter mode employs a unique combination of signal sources for the Clock, Direction and Index signals. To standardize the descriptions of counter operating modes, the following symbolic names have been assigned to the various counter input signal sources:

Signal Name	Function
`SYS_C`	Fixed frequency, 2.0 MHz clock.
`ENC_C`	Encoder clock input, B-phase.
`ENC_D`	Encoder clock input, A-phase.
`ENC_X`	Encoder index input.
`OVR_A`	Overflow output from associated A counter.

4.6.1.2 Using DIOs as Counter Outputs

DIO channels 0 through 5 may be configured to function as counter overflow outputs. See “S626_DIOModeSet()” and “S626_DIOModeGet()” on page 19 for details.

4.6.1.3 Mode Descriptions

Counter operating modes are listed in Table 9. These modes are selected by means of the `S626_CounterModeSet()` function, as described in Section 4.6.3.

Table 8: Symbols used to describe programmable attributes.

Symbol	Meaning
✍	Attribute is programmable.
✍	Attribute is not programmable.
1	Clock multiplier `1x` is supported.
1, 2, 4	Clock multipliers `1x`, `2x` and `4x` are supported.

For each mode, Table 9 lists the input signal sources (detailed in Section 4.6.1.1) and all supported programmable attributes. Symbols used to describe the programmable attributes are listed in Table 8.

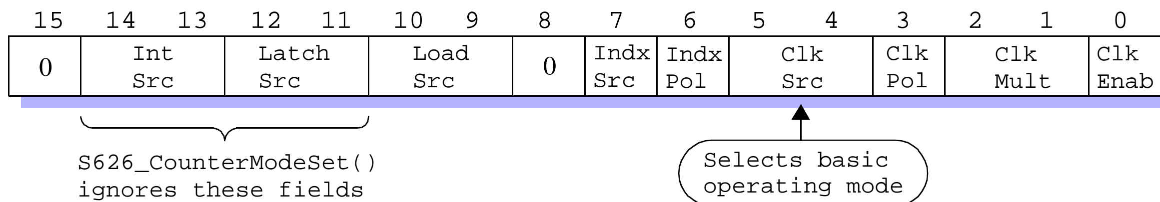
Table 9: Counter operating modes.

Mode	Input Signal Sources		Programmable Attributes		Applies To Counters	Description
	Clock	Direction	Clock Polarity	Clock Multiplier		
Timer	`SYS_C`	`Fixed`	✍	**1**	**A and B**	Fixed 2 MHz Clock.
Counter	`ENC_C/ENC_D`		✍	**1, 2, 4**	**A and B**	Clock and direction driven by external inputs.
Extender	`OVR_A`	`ENC_D`	✍	**1**	**B only**	Clock driven by overflow output from paired A counter, direction driven by external input.

4.6.2 Configuration Options

A counter's operational configuration is programmed by means of `S626_CounterModeSet()` and may be retrieved via `S626_CounterModeGet()`. These two functions employ a data structure, named "COUNTER_SETUP," to convey all counter configuration parameters. The COUNTER_SETUP structure is a WORD value that contains a collection of bit fields, as shown in Figure 3.

Figure 3: Bit fields in the COUNTER_SETUP structure.



Except for the `IntSrc` and `LatchSrc` bit fields, all of the bit fields in the COUNTER_SETUP structure are used by both `S626_CounterModeSet()` and `S626_CounterModeGet()`. The `IntSrc` and `LatchSrc` bit fields are utilized only by `S626_CounterModeGet()`; they are ignored by the `S626_CounterModeSet()` function.

Bit fields that contain "0" are not used. To ensure compatibility with future driver enhancements, however, applications should be designed so as to ignore these fields.

The functions of the COUNTER_SETUP bit fields are described in the following sections. Note that the fundamental operating mode—Timer, Counter or Extender—is determined by the `ClkSrc` bit field, as described in Section 4.6.2.6.

4.6.2.1 IntSrc

`IntSrc` specifies the captured events that will cause an interrupt request.

The `IntSrc` bit field is ignored by `S626_CounterModeSet()`. When invoked, `S626_CounterModeSet()` resets all captured events on the target counter and disables the counter's interrupts by forcing `IntSrc` to `INTSRC_NONE`. After configuring a counter, `S626_CounterIntSourceSet()` may be called to enable the counter to generate interrupts.

IntSrc may have one of the following values:

Value	Symbolic Name	Events that will generate interrupts.
0	INTSRC_NONE	None. Interrupts are disabled.
1	INTSRC_OVER	Captured counter overflow.
2	INTSRC_INDX	Captured counter index.
3	INTSRC_ANY	Captured counter overflow or index.

4.6.2.2 LatchSrc

LatchSrc specifies the events that will cause the counter contents to be latched. This attribute is shared by the A and B counters belonging to a counter pair.

This bit field is ignored by S626_CounterModeSet(). When invoked, S626_CounterModeSet() will preserve the current state of LatchSrc on the target counter. You must use S626_CounterLatchSourceSet() to modify the value of LatchSrc for the target counter.

LatchSrc may have one of the following values:

Value	Symbolic Name	Trigger Condition
0	LATCHSRC_AB_READ	Counter A or B latched when read by S626_CounterReadLatch().
1	LATCHSRC_A_INDXA	Counter A latched in response to counter A index.
2	LATCHSRC_B_INDXB	Counter B latched in response to counter B index.
3	LATCHSRC_B_OVERA	Counter B latched in response to counter A overflow.

4.6.2.3 LoadSrc

LoadSrc specifies the trigger event that will cause the preload register to be transferred into the counter core. This event may be selected by means of the LoadSrc bit in conjunction with S626_CounterModeSet(), or alternatively, by calling S626_CounterLoadTrig().

The meaning of LoadSrc bit field depends on whether an A or B counter is being addressed. In the case of A counters, LoadSrc may be set to one of the following values:

Value	Symbolic Name	Trigger Event
0	LOADSRC_INDX	Counter A index.
1	LOADSRC_OVER	Counter A overflow.
2	LOADSRCA_NONE	Disabled.
3	LOADSRC_NONE	Disabled.

In the case of B counters, LoadSrc may be set to one of the following values:

Value	Symbolic Name	Trigger Event
0	LOADSRC_INDX	Counter B index.

Value	Symbolic Name	Trigger Event
1	LOADSRC_OVER	Counter B overflow.
2	LOADSRCB_OVERA	Paired counter A overflow. This option is typically used in frequency counter applications in which the A counter (configured as a Timer) provides measurement interval control and the paired B counter (configured as a Counter) accumulates pulses from the frequency source. Each time the A counter overflows, the B counter is automatically latched and reset to zero in preparation for the next measurement interval.
3	LOADSRC_NONE	Disabled.

4.6.2.4 IndxSrc

IndxSrc specifies the signal source for the counter's index input.

IndxSrc may be set to one of the following values:

Value	Symbolic Name	Description
0	INDXSRC_HARD	Counter index is driven by the encoder physical index input, ENC_X, or by software control via S626_CounterSoftIndex().
1	INDXSRC_SOFT	Counter index is controlled only by software via S626_CounterSoftIndex(). The physical encoder index input, ENC_X, is ignored.

4.6.2.5 IndxPol

IndexPol specifies the index edge to be used for triggering index-driven events. IndexPol is forced to zero if IndxSrc is set to INDXSRC_SOFT.

IndexPol may be set to one of the following values:

Value	Symbolic Name	Description
0	INDXPOL_POS	Selects the rising (inactive-to-active) edge transition.
1	INDXPOL_NEG	Selects the falling (active-to-inactive) edge transition. This will be ignored and IndexPol will be set to Rising if the IndxSrc bit field is set to INDXSRC_SOFT.

4.6.2.6 ClkSrc

ClkSrc specifies the signal source to be used for clocking the counter core. In addition, this bit field implicitly determines the basic operating mode for the counter channel.

ClkSrc may be set to one of the following values:

Value	Symbolic Name	Description
0-1	CLKSRC_COUNTER	Applicable to A and B counters. Selects ENC_C and ENC_D as clock signal sources. Quadrature-encoded, two-phase clocks are connected to ENC_C and ENC_D. In the case of single-phase clock sources, the clock signal connects to ENC_C and the direction control signal connects to ENC_D.
2	CLKSRC_TIMER	Applicable to A and B counters. Selects the 2 MHz system clock as the clock source. Count direction is controlled by the ClkPol bit field.
3	CLKSRC_EXTENDER	Applicable to B counters only. Selects the overflow output from the paired A counter as the clock source, and ENC_D as the direction control signal. If this is specified for an A counter when calling S626_CounterModeSet(), the configuration request will be rejected and the counter will not be configured to the new settings.

4.6.2.7 ClkPol

`ClkPol` has a functionality that is determined by the counter operating mode (which is implicitly specified by `ClkSrc`). In the Counter and Extender modes, `ClkPol` specifies the clock edge to be used for clocking the counter core. In the Timer mode, `ClkPol` specifies the count direction.

When the counter is operating in Timer mode, `ClkPol` may be set to one of the following values:

Value	Symbolic Name	Description
0	CNTDIR_UP	Programs the counter to count up.
1	CNTDIR_DOWN	Programs the counter to count down.

When the counter is operating in Counter or Extender mode, `ClkPol` may be set to one of the following values:

Value	Symbolic Name	Description
0	CLKPOL_POS	Selects the rising (inactive-to-active) clock edge transition.
1	CLKPOL_NEG	Selects the falling (active-to-inactive) clock edge transition.

4.6.2.8 ClkMult

`ClkMult` specifies whether the frequency of the clock source (specified by `ClkSrc`) will be multiplied by 1, 2 or 4 before being applied to the counter core.

The `1x` multiplier is legal in all operating modes, and the `2x` and `4x` multipliers are valid only in the Counter mode. The `1x` multiplier will be programmed by default if a multiplier is specified that is invalid for the specified operating mode. Although they are valid in the Counter mode, the `2x` and `4x` multipliers should not be used with counters that are driven by single-phase clock sources as this will cause unpredictable behavior.

`ClkMult` may be set to one of the following values:

Value	Symbolic Name	Description
0	CLKMULT_4X	Valid only in Counter mode. Counter will count in response to any edge that occurs on either of the quadrature clock inputs. Use this multiplier only with two-phase, quadrature-encoded clock sources.
1	CLKMULT_2X	Valid only in Counter mode. Counter will count in response to any edge that occurs on the phase B clock input. Use this multiplier only with two-phase, quadrature-encoded clock sources.
2-3	CLKMULT_1X	Valid in all modes. Counter will count in response to an inactive-to-active edge on the selected clock input.

4.6.2.9 ClkEnab

`ClkEnab` specifies the conditions that enable counting to occur. Count enabling conditions may be programmed by means of the `ClkEnab` bit in conjunction with `S626_CounterModeSet()`, or alternatively, via `S626_CounterEnableSet()`.

`ClkEnab` may be set to one of the following values:

Value	Symbolic Name	Description
0	CLKENAB_ALWAYS	Counting is always enabled, regardless of the index state.
1	CLKENAB_INDEX	Counting is enabled only when the index is active.

4.6.3 S626_CounterModeSet()

Function: Programs a counter's operational configuration.

Prototype: `VOID S626_CounterModeSet( HBD hbd, WORD chan, WORD options );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`chan`	`WORD`	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
`options`	`WORD`	Configuration options, organized as a COUNTER_SETUP structure. See Section 4.6.2 for details.

Returns: None.

Notes: Always call this function first to establish a counter's operating configuration before attempting to modify any independently programmable attributes. This is necessary because `S626_CounterModeSet()` initializes most of the counter configuration attributes to either their default values or to the values specified by `options`.

This function clears any captured index or overflow events on the target counter channel and sets the counter interrupt source to `None`, thereby disabling interrupts. Out of all the remaining programmable counter configuration options, only the latch source (Section 4.6.8), which is shared by the A and B counters of the associated counter pair, is left unmodified.

`S626_CounterModeSet()` will reject the configuration request and preserve the existing counter configuration if any illegal options are specified. After calling this function, `S626_CounterModeGet()` may be called to determine whether the counter was successfully configured.

Example: See "Counter Programming Examples" on page 38.

4.6.4 S626_CounterModeGet()

Function: Returns a counter's operational configuration.

Prototype: `WORD S626_CounterModeGet( HBD hbd, WORD chan );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`chan`	`WORD`	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: WORD value, organized as a COUNTER_SETUP structure. See Section 4.6.2 for details.

Notes: `S626_CounterModeGet()` provides a mechanism for retrieving programmed counter configuration attributes. This can be useful in several situations:

- Confirms that a counter was successfully configured by `S626_CounterModeSet()`.
- In battery-backed applications, this function can be used to determine the current counter configurations following a power failure, thereby enabling counter operations to resume after a system power failure has occurred.
- Serves as a diagnostic tool for verifying hardware operation during application development.
- Provides support for application run-time diagnostics.
- Eliminates the need for applications to maintain storage for counter configuration information.

Example:

```
////////////////////////////////////////////////////////////////////////
// Display the current clock multiplier for board 0, counter 2B.
////////////////////////////////////////////////////////////////////////

switch ( ( S626_CounterModeGet( 0, 5 ) >> 1 ) & 3 )
{
    case 0: printf( "Mult is 4x.\n" ); break;
    case 1: printf( "Mult is 2x.\n" ); break;
    case 2: printf( "Mult is 1x.\n" ); break;
}
```

4.6.5 S626_CounterEnableSet()

Function: Specifies the conditions that enable counting to occur.

Prototype: `VOID S626_CounterEnableSet( HBD hbd, WORD chan, WORD cond );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`chan`	`WORD`	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
`cond`	`WORD`	Specifies the conditions that enable counting to occur. See below for details.

The `cond` argument may be set to one of the following values:

Value	Symbolic Name	Description
`0`	`CLKENAB_ALWAYS`	Counting is always enabled, regardless of the index state.
`1-65535`	`CLKENAB_INDEX`	Counting is enabled only when the index is active.

Returns: None.

Notes: Count enabling conditions may also be programmed by means of `S626_CounterModeSet()`. The `S626_CounterModeGet()` function can be used to determine the count enabling conditions currently in effect.

Example: See “Counter Programming Examples” on page 38.

4.6.6 S626_CounterPreload()

Function: Writes a value to a counter’s preload register.

Prototype: `VOID S626_CounterPreload( HBD hbd, WORD chan, DWORD value );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`chan`	`WORD`	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
`value`	`DWORD`	Value to be written to the preload register. Only the least significant 24 bits of this value are written to the preload register; the high byte is ignored.

Returns: None.

Notes: This function stores a value in a counter's preload register, but it does *not* transfer the value to the counter core. The preload register is transferred to the counter core only in response to a predefined Load Trigger event. See Section 4.6.7 for details.

Example: See "Counter Programming Examples" on page 38.

4.6.7 S626_CounterLoadTrigSet()

Function: Selects the events that will cause a counter's preload register to be transferred to its counter core.

Prototype: `VOID S626_CounterLoadTrigSet( HBD hbd, WORD chan, WORD events );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`chan`	`WORD`	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
`events`	`WORD`	Indicates the events that trigger a transfer of data from the preload register to the counter core. See below for details.

The `events` argument specifies the event or events that will trigger the transfer of data from the preload register into the counter core. The meaning of `events` depends on whether an A or B counter is being addressed.

In the case of A counters, `events` may be set to one of the following values:

Value	Symbolic Name	Trigger Event
0	LOADSRC_INDX	Counter A index.
1	LOADSRC_OVER	Counter A overflow.
2	LOADSRCA_NONE	Disabled.
3	LOADSRC_NONE	Disabled.
4-65535	---	Illegal. This will cause the ERR_ILLEGAL_PARM error flag to be set.

In the case of B counters, `events` may be set to one of the following values:

Value	Symbolic Name	Trigger Event
0	LOADSRC_INDX	Counter B index.
1	LOADSRC_OVER	Counter B overflow.
2	LOADSRCB_OVERA	Counter A overflow.
3	LOADSRC_NONE	Disabled.
4-65535	---	Illegal. This will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: None.

Notes: Preload trigger events may also be programmed by means of `S626_CounterModeSet()`. The `S626_CounterModeGet()` function can be used to determine the preload trigger events currently in effect.

Example: See "Counter Programming Examples" on page 38.

4.6.8 S626_CounterLatchSourceSet()

Function: Selects the events that will cause a counter's contents to be transferred to its latch register.

Prototype: `VOID S626_CounterLatchSourceSet( HBD hbd, WORD chan, WORD events );`

Parameter	Type	Description
hbd	HBD	Board Handle.
chan	WORD	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
events	WORD	Indicates the events that trigger a transfer of data from the counter core to the latch register. See below for details.

The `events` argument specifies which events will act as triggers for transferring the counts from the counter core into the latch register. This argument may be set to one of the following values:

Value	Symbolic Name	Description
0	LATCHSRC_AB_READ	Counter A or B latched when read by `S626_CounterReadLatch()`.
1	LATCHSRC_A_INDXA	Counter A latched in response to counter A index.
2	LATCHSRC_B_INDXB	Counter B latched in response to counter B index.
3	LATCHSRC_B_OVERA	Counter B latched in response to counter A overflow.
4-65535	---	Illegal. This will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: None.

Notes: This function selects the latch trigger for both the A and B counters of a counter pair. When the latch trigger is programmed for an A (or B) counter, the paired B (or A) counter is also programmed at the same time.

Example: See "Counter Programming Examples" on page 38.

4.6.9 S626_CounterReadLatch()

Function: Read a counter's latch register.

Prototype: `DWORD S626_CounterReadLatch( HBD hbd, WORD chan );`

Parameter	Type	Description
hbd	HBD	Board Handle.
chan	WORD	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: DWORD consisting of the value stored in the counter latch register. The returned number will have a value between `0x00000000` and `0x00FFFFFF`, inclusive.

Notes: Each counter latch register is shared by both the A and B counter belonging to a counter pair. Depending on the programming of the counter latch trigger source, various events will cause the counter core to be transferred to the counter latch register. See Section 4.6.8 for details.

Example: See "Counter Programming Examples" on page 38.

4.6.10 S626_CounterCapStatus()

Function: Returns event capture flags for all counter channels.

Prototype: `WORD S626_CounterCapStatus( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD value containing Overflow and Index capture flags for all counter channels. This value is structured as a set of bit fields as shown in Figure 4.

Figure 4: WORD value returned by S626_CounterCapStatus().

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Over 2B	Over 2A	Over 1B	Over 1A	Over 0B	Over 0A	Indx 2B	Indx 2A	Indx 1B	Indx 1A	Indx 0B	Indx 0A	0	0	0	0

Each capture flag occupies a one-bit field in the returned WORD value. A logic one in a bit field indicates that the associated event is captured, while a zero in the bit field indicates that the associated event has not been captured. For example, a logic one in bit 7 indicates that an Index edge was detected on counter 1B.

Notes: The four least-significant bits of the returned value is always zero. This enables rapid testing for captured events on all counters.

Counter event capturing is always enabled. A counter's event capture flags are reset to zero in response to any of the following actions:

- `S626_CounterModeSet()` is called to set the counter's operating mode.
- `S626_CounterCapFlagsReset()` is called to explicitly reset the counter's capture flags.
- `S626_CounterIntSourceSet()` is called to select the counter's interrupt source.

Example: See "Counter Programming Examples" on page 38.

4.6.11 S626_CounterCapFlagsReset()

Function: Resets a counter's event capture flags.

Prototype: `VOID S626_CounterCapFlagsReset( HBD hbd, WORD chan );`

Parameter	Type	Description
hbd	HBD	Board Handle.
chan	WORD	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: None.

Notes: This function resets the Overflow and Index event capture flags associated with the target counter (see Figure 4). In cases where either (or both) of these events flags are selected as interrupt sources, this function will negate the pending interrupt service request.

Example: See "Counter Programming Examples" on page 38, and "Example: Counter Interrupts" on page 52.

4.6.12 S626_CounterSoftIndex()

Function: Toggles a counter's index.

Prototype: `VOID S626_CounterSoftIndex( HBD hbd, WORD chan );`

Parameter	Type	Description
hbd	HBD	Board Handle.
chan	WORD	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: None.

Notes: `S626_CounterSoftIndex()` briefly inverts a counter channel's index so as to produce an index pulse. This has the effect of simulating a hardware pulse on the currently selected index input. This can be useful for triggering index-controlled actions, such as counter preloading or latching, under software control.

This function does not physically drive the selected index input signal. Instead, it modifies an internal, buffered image of the applied physical input.

Example: See "Counter Programming Examples" on page 38.

4.6.13 S626_CounterIntSourceSet()

Function: Specifies the captured events that are permitted to cause interrupts.

Prototype: `VOID S626_CounterIntSourceSet( HBD hbd, WORD chan, WORD events );`

Parameter	Type	Description
hbd	HBD	Board Handle.
chan	WORD	Counter channel number. Legal values are 0 through 5, as described in Section 3.3. Specifying a channel number greater than 5 will cause the ERR_ILLEGAL_PARM error flag to be set.
events	WORD	Specifies the events that will cause interrupts. See below for details.

The `events` parameter may be set to one of the following values:

Value	Symbolic Name	Captured events that will cause interrupts.
0	INTSRC_NONE	None. Interrupts will be disabled.
1	INTSRC_OVER	Counter overflow.
2	INTSRC_INDX	Counter index.
3	INTSRC_ANY	Counter overflow or index.
4-65535	---	Illegal. This will cause the ERR_ILLEGAL_PARM error flag to be set.

Returns: None.

Notes: In addition to selecting the events that can cause interrupts, this function also resets the Index and Overflow capture flags for the target counter channel. This prevents interrupts from occurring immediately in case interrupts are being enabled for events that have already been captured.

To enable counter interrupts, this function must be called after `S626_CounterModeSet()` has been called, as `S626_CounterModeSet()` automatically disables all interrupts for the target counter channel.

Example: See Section 4.7, and "Example: Counter Interrupts" on page 52

4.7 Counter Programming Examples

4.7.1 Constants Used in Examples

Various numerical constants are referenced by the counter programming examples. Instead of duplicating constants in each example, all of the constants have been gathered here as they would be in an application include file. The following source code defines most of the constants used in the programming examples.

```
////////////////////////////////////////////////////////////////////////////////////
//  Numerical constants used by the counter programming examples.
////////////////////////////////////////////////////////////////////////////////////

                                            // LoadSrc values:
#define LOADSRC_INDX        0               //   Preload core in response to Index.
#define LOADSRC_OVER        1               //   Preload core in response to Overflow.
#define LOADSRCB_OVERA      2               //   Preload B core in response to A Overflow.
#define LOADSRC_NONE        3               //   Never preload core.
                                            // IntSrc values:
#define INTSRC_NONE         0               //   Interrupts disabled.
#define INTSRC_OVER         1               //   Interrupt on Overflow.
#define INTSRC_INDX         2               //   Interrupt on Index.
#define INTSRC_BOTH         3               //   Interrupt on Index or Overflow.
                                            // LatchSrc values:
#define LATCHSRC_AB_READ    0               //   Latch on read.
#define LATCHSRC_A_INDXA    1               //   Latch A on A Index.
#define LATCHSRC_B_INDXB    2               //   Latch B on B Index.
#define LATCHSRC_B_OVERA    3               //   Latch B on A Overflow.
                                            // IndxSrc values:
#define INDXSRC_HARD        0               //   Hardware or software index.
#define INDXSRC_SOFT        1               //   Software index only.
                                            // IndxPol values:
#define INDXPOL_POS         0               //   Index input is active high.
#define INDXPOL_NEG         1               //   Index input is active low.
                                            // ClkSrc values:
#define CLKSRC_COUNTER      0               //   Counter mode.
#define CLKSRC_TIMER        2               //   Timer mode.
#define CLKSRC_EXTENDER     3               //   Extender mode.
                                            // ClkPol values:
#define CLKPOL_POS          0               //   Counter/Extender clock is active high.
#define CLKPOL_NEG          1               //   Counter/Extender clock is active low.
#define CNTDIR_UP           0               //   Timer counts up.
#define CNTDIR_DOWN         1               //   Timer counts down.
                                            // ClkEnab values:
#define CLKENAB_ALWAYS      0               //   Clock always enabled.
#define CLKENAB_INDEX       1               //   Clock is enabled by index.
                                            // ClkMult values:
#define CLKMULT_4X          0               //   4x clock multiplier.
#define CLKMULT_2X          1               //   2x clock multiplier.
#define CLKMULT_1X          2               //   1x clock multiplier.
                                            // Bit Field positions in COUNTER_SETUP word:
#define BF_LOADSRC          9               //   Preload trigger.
#define BF_INDXSRC          7               //   Index source.
#define BF_INDXPOL          6               //   Index polarity.
#define BF_CLKSRC           4               //   Clock source.
#define BF_CLKPOL           3               //   Clock polarity/count direction.
#define BF_CLKMULT          1               //   Clock multiplier.
#define BF_CLKENAB          0               //   Clock enable.
                                            // Counter channel numbers:
```

```
#define CNTR_0A          0          //   Counter 0A.
#define CNTR_1A          1          //   Counter 1A.
#define CNTR_2A          2          //   Counter 2A.
#define CNTR_0B          3          //   Counter 0B.
#define CNTR_1B          4          //   Counter 1B.
#define CNTR_2B          5          //   Counter 2B.

// In addition to the constants listed above, the following code macros are useful.
// Counter overflow/index event flag bit masks for S626_CounterCapStatus():
#define INDXMASK(C)    ( 1 << ( ( (C) > 2 ) ? ( (C) * 2 - 1 ) : ( (C) * 2 +  4 ) ) )
#define OVERMASK(C)    ( 1 << ( ( (C) > 2 ) ? ( (C) * 2 + 5 ) : ( (C) * 2 + 10 ) ) )
```

4.7.2 Example: Periodic Interrupt Generator

This example shows how to program a counter to generate interrupts at periodic intervals.

A function, `GenPeriodicInts()`, encapsulates the complexities of configuration programming and provides a standardized method that works with any counter. This function will configure any of the six counter channels to generate interrupts at intervals specified by the `milliseconds` argument.

The target counter is configured as a down-counting timer. Clock enabling is controlled by the index, which in turn is controlled exclusively by software; the hardware index input is disabled. When the counter counts down to zero, it automatically reloads from the preload register and generates an interrupt service request via the counter overflow event capture.

```
///////////////////////////////////////////////////////////////////////////////////
// Configure a counter to generate periodic interrupts.
///////////////////////////////////////////////////////////////////////////////////

VOID GenPeriodicInts( HBD hbd, WORD chan, DWORD milliseconds )
{
    // Set counter operating mode.
    S626_CounterModeSet( hbd, chan,
        ( LOADSRC_INDX  << BF_LOADSRC ) |      // Index causes preload.
        ( INDXSRC_SOFT  << BF_INDXSRC ) |      // Hardware index disabled.
        ( CLKSRC_TIMER  << BF_CLKSRC  ) |      // Operating mode is Timer.
        ( CNTDIR_DOWN   << BF_CLKPOL  ) |      // Count direction is Down.
        ( CLKMULT_1X    << BF_CLKMULT ) |      // Clock multiplier is 1x.
        ( CLKENAB_INDEX << BF_CLKENAB ) );     // Counting is initially disabled.

    // Initialize PreLoad value to match the specified time interval.  Since the counter
    // clock is fixed at 2 MHz, this is computed by multiplying milliseconds by 2,000.
    S626_CounterPreload( hbd, chan, milliseconds * 2000 );

    // Generate a soft index to force transfer of PreLoad value into counter core.
    S626_CounterSoftIndex( hbd, chan );

    // Enable transfer of PreLoad value to counter in response to overflow.  This
    // will cause the initial counts to reload every time the counts reach zero.
    S626_CounterLoadTrigSet( hbd, chan, LOADSRC_OVER  );

    // Enable the counter to generate interrupt requests upon captured overflow.
    S626_CounterIntSourceSet( hbd, chan, INTSRC_OVER );

    // Enable the timer.  The first interrupt will occur after the specified
    // time interval elapses.
    S626_CounterEnableSet( hbd, chan, CLKENAB_ALWAYS );
}
```

This code shows how to use `GenPeriodicInts()` to activate periodic interrupts:

```
// Configure board 0, counter 0A to generate an interrupt every 50 milliseconds.
GenPeriodicInts( 0, CNTR_0A, 50 );
```

4.7.3 Example: Encoder Interface

This example shows how to program a counter to work with an incremental encoder.

Incremental encoders produce two clock signals that have a 90 degree phase difference (i.e., *quadrature-encoded*), from which the counter core derives its count clock and direction control signals. Often, encoder applications also utilize an index signal that is used for registering a reference position.

In this example, the counter clock inputs is driven by the encoder's quadrature clock. The encoder index input is programmed to reset the counter to zero and source an interrupt request when the index switches to its active state. The counter is configured so that encoder counts may be read by the application at any time.

```
/////////////////////////////////////////////////////////////////////////////////////////
// Configure board 0, counter 2A as a quadrature counter.  An active Index will reset
// the counter core to zero and generate an interrupt request.
/////////////////////////////////////////////////////////////////////////////////////////

// Set counter operating mode.
S626_CounterModeSet( 0, CNTR_2A,
    ( LOADSRC_INDX   << BF_LOADSRC ) |     // Index causes preload.
    ( INDXSRC_HARD   << BF_INDXSRC ) |     // Hardware index is enabled.
    ( INDXPOL_POS    << BF_INDXPOL ) |     // Active high index.
    ( CLKSRC_COUNTER << BF_CLKSRC  ) |     // Operating mode is Counter.
    ( CLKPOL_POS     << BF_CLKPOL  ) |     // Active high clock.
    ( CLKMULT_4X     << BF_CLKMULT ) |     // Clock multiplier is 4x.
    ( CLKENAB_ALWAYS << BF_CLKENAB ) );    // Counting is always enabled.

// Initialize preload value to zero so that the counter core will be set
// to zero upon the occurance of an Index.
S626_CounterPreload( 0, CNTR_2A, 0 );

// Enable latching of accumulated counts on demand.  This assumes that
// there is no conflict with the latch source used by paired counter 2B.
S626_CounterLatchSourceSet( 0, CNTR_2A, LATCHSRC_AB_READ );

// Enable the counter to generate interrupt requests upon index.
S626_CounterIntSourceSet( 0, CNTR_2A, INTSRC_INDX );
```

As shown in below, the accumulated encoder counts may be read at any time. To ensure that a legitimate counts value will be displayed, at least one index must occur before executing this statement.

```
// Read and display current encoder position.
printf( "Encoder position = %d\n", S626_CounterReadLatch( 0, CNTR_2A ) );
```

4.7.4 Example: Simple Event Counter

This example shows how to program a counter to count pulses from a single-phase clock source, such as a tachometer.

In this example, the counter clock input is driven by an external, single-phase signal source and the count direction is fixed so that the counter core increments once per input pulse. The counter's hardware index input is disabled so that the index is exclusively software-driven. The counter is configured so that event counts may be read by the application at any time.

```
/////////////////////////////////////////////////////////////////////////////////////////
// Configure board 0, counter 2A as an event counter.
/////////////////////////////////////////////////////////////////////////////////////////
```

```
// Set counter operating mode.
S626_CounterModeSet( 0, CNTR_2A,
    ( LOADSRC_INDX    << BF_LOADSRC ) |      // Preload in response to index.
    ( INDXSRC_SOFT    << BF_INDXSRC ) |      // Hardware index disabled.
    ( CLKSRC_COUNTER << BF_CLKSRC   ) |      // Operating mode is Counter.
    ( CLKPOL_POS      << BF_CLKPOL  ) |      // Active high clock.
    ( CLKMULT_1X      << BF_CLKMULT ) |      // Clock multiplier is 1x.
    ( CLKENAB_ALWAYS << BF_CLKENAB ) );      // Counting is always enabled.

// Initialize preload value to zero so that the counter core will be set
// to zero upon the occurance of a software-induced index.
S626_CounterPreload( 0, CNTR_2A, 0 );

// Enable latching of accumulated counts on demand.  This assumes that
// there is no conflict with the latch source used by paired counter 2B.
S626_CounterLatchSourceSet( 0, CNTR_2A, LATCHSRC_AB_READ );

// Generate a soft index to initialize the counts to zero.
S626_CounterSoftIndex( hbd, CNTR_2A );
```

The accumulated event count may be read by executing the following statement:

```
// Read and display the accumulated event counts.
printf( "Events counted = %d\n", S626_CounterReadLatch( 0, CNTR_2A ) );
```

Counts may be reset to zero by executing the following statement:

```
// Reset the accumulated counts.
S626_CounterSoftIndex( hbd, CNTR_2A );
```

4.7.5 Example: Pulse Width Measurement

This example shows how to use a counter to measure pulse width. In this example, the counter is configured to measure positive pulses, although it is a simple matter to reconfigure it to measure negative pulses.

The counter is configured as a timer with a hardware-gated clock. The counter's hardware index input, which connects to the signal that is to be measured, gates the counter's clock so that counting is enabled only during input pulses. In addition to the level-sensitive clock gating function, the index input also performs the following edge-sensitive functions in response to an input pulse leading edge:

- Transfers the previously acquired pulse width data to the latch register.
- Preloads the counter core with zeros to begin the new acquisition.
- Captures a counter Index Event.
- Generates an interrupt request.

```
//////////////////////////////////////////////////////////////////////
//  Measure positive pulse width using board 0, counter 2A.
//////////////////////////////////////////////////////////////////////

// Set counter operating mode.
S626_CounterModeSet( 0, CNTR_2A,
    ( LOADSRC_INDX   << BF_LOADSRC ) |      // Preload in response to index.
    ( INDXSRC_HARD   << BF_INDXSRC ) |      // Pulse signal drives index.
    ( INDXPOL_POS    << BF_INDXPOL ) |      // Active high pulse signal.
    ( CLKSRC_TIMER   << BF_CLKSRC  ) |      // Operating mode is Timer.
    ( CNTDIR_UP      << BF_CLKPOL  ) |      // Count direction is Up.
    ( CLKMULT_1X     << BF_CLKMULT ) |      // Clock multiplier is 1x.
    ( CLKENAB_INDEX << BF_CLKENAB ) );      // Counting is gated by index.
```

```
// Initialize preload value to zero so that the counter core will be set
// to zero upon the occurance of a hardware index.
S626_CounterPreload( 0, CNTR_2A, 0 );

// Enable latching of accumulated counts in response to an index.  This assumes that
// there is no conflict with the latch source used by paired counter 2B.
S626_CounterLatchSourceSet( 0, CNTR_2A, LATCHSRC_A_INDXA );
```

After configuring the counter, the application waits for an index event. For the sake of clarity, the index event is detected by means of a polling loop, whereas in a real application this would more likely be implemented with interrupts. When an index event is captured, the acquired pulse width data is fetched from the counter's latch register. Data should be discarded for the first index event because valid data is not yet available from a previous acquisition. The following code shows the polling loop.

```
// Ignore the data value from the first event.
bool IsFirst = true;

for ( ; ; )
{
    // Wait for a captured index event on counter 2A, then clear its capture flag.
    while ( !( S626_CounterCapStatus( 0 ) & INDXMASK( CNTR_2A ) ) );
    S626_CounterCapFlagsReset( 0, CNTR_2A );

    // Read and display measured pulse width.  Since timer is clocked by a 2 MHz source,
    // accumulated count is divided by 2 to convert pulse width to microseconds units.
    if ( !IsFirst )
        printf( "Pulse width (µs) = %d\n", S626_CounterReadLatch( 0, CNTR_2A ) >> 1 );

    IsFirst = false;
}
```

4.7.6 Example: Frequency Counter

This example shows how to use a pair of counters to measure frequency.

A function, `CreateFreqCounter()`, is implemented to encapsulate the complexities of configuration programming and to provide a standardized method that works with any counter pair. This function will configure any of the three counter channel pairs to measure frequency.

CounterA serves as an acquisition gate generator. The gate time is determined by the value in the counterA preload register. The end of the gate interval is denoted by an overflow on counterA.

CounterB's clock input is connected to the external frequency source; this counter continuously counts pulses from the frequency source. Upon overflow of counterA, counterB transfers its counts to the latch register and resets to zero to begin the next acquisition.

```
/////////////////////////////////////////////////////////////////////
// Configure an A/B counter pair to measure frequency.
// The counter pair is specified by the A member of the pair.
// Acquisition gate time is specified in milliseconds.
/////////////////////////////////////////////////////////////////////

VOID CreateFreqCounter( HBD hbd, WORD CounterA, WORD GateTime )
{
    // Set operating mode for counterA.
    S626_CounterModeSet( hbd, CounterA,
        ( LOADSRC_OVER  << BF_LOADSRC ) |      // Preload upon overflow.
        ( INDXSRC_SOFT  << BF_INDXSRC ) |      // Disable hardware index.
        ( CLKSRC_TIMER  << BF_CLKSRC  ) |      // Operating mode is Timer.
        ( CNTDIR_DOWN   << BF_CLKPOL  ) |      // Count direction is Down.
        ( CLKMULT_1X    << BF_CLKMULT ) |      // Clock multiplier is 1x.
        ( CLKENAB_INDEX << BF_CLKENAB ) );     // Counting is initially disabled.
```

```
    // Set counterA core and preload value to the desired gate time.  Since the counter
    // clock is fixed at 2 MHz, this is computed by multiplying milliseconds by 2,000.
    S626_CounterPreload( hbd, CounterA, GateTime * 2000 );
    S626_CounterSoftIndex( hbd, CounterA );

    // Enable preload of counterA in response to overflow.  This causes the timer to
    // restart automatically when its counts reach zero.
    S626_CounterLoadTrigSet( hbd, CounterA, LOADSRC_OVER  );

    // Set operating mode for counterB.
    S626_CounterModeSet( 0, CounterA + 3,
        ( LOADSRCB_OVERA << BF_LOADSRC ) |      // Preload zeros upon leading gate edge.
        ( INDXSRC_SOFT   << BF_INDXSRC ) |      // Hardware index is disabled.
        ( CLKSRC_COUNTER << BF_CLKSRC  ) |      // Operating mode is Counter.
        ( CLKPOL_POS     << BF_CLKPOL  ) |      // Clock is active high.
        ( CLKMULT_1X     << BF_CLKMULT ) |      // Clock multiplier is 1x.
        ( CLKENAB_ALWAYS << BF_CLKENAB ) );     // Clock is always enabled.

    // Initialize counterB's preload value to zero so that counterB core will be set
    // to zero in response to trailing gate edge (counter A overflow).
    S626_CounterPreload( 0, CounterA + 3, 0 );

    // Enable latching of counterB's acquired frequency data in response to trailing
    // gate edge (counterA overflow).
    S626_CounterLatchSourceSet( 0, CounterA + 3, LATCHSRC_B_OVERA );

    // Enable the acquisition gate generator.
    S626_CounterEnableSet( hbd, CounterA, CLKENAB_ALWAYS );
}
```

After configuring the counter pair, the application waits for an overflow event on counter A. For the sake of clarity, the overflow event is detected by means of a polling loop, whereas in a real application this would more likely be implemented with interrupts (see Section 4.11.2 for an implementation that uses interrupts).

When a counter A overflow event is captured, the acquired frequency data is fetched from the latch register. Data should be discarded for the first overflow event because data is not yet available from a previous acquisition. The following code shows the counter initialization and polling loop.

```
#define COUNTER   CNTR_0A    // Counter A (and implicitly, B) channel to use.

// Configure counter A/B pair as a frequency counter, gate time = 1 second.
CreateFreqCounter( 0, COUNTER, 1000 );

// Ignore the data value from the first gate period.
bool IsFirst = true;

for ( ; ; )
{
    // Wait for a captured overflow event on counter A, then clear its capture flag.
    while ( !( S626_CounterCapStatus( 0 ) & OVERMASK( COUNTER ) ) );
    S626_CounterCapFlagsReset( 0, COUNTER );

    // Read and display measured frequency from counter B.
    if ( !IsFirst )
        printf( "Frequency (Hz) = %8d.\r", S626_CounterReadLatch( 0, COUNTER + 3 ) );

    IsFirst = false;
}
```

4.8 Watchdog Timer Functions

4.8.1 S626_WatchdogPeriodSet()

Function: Programs the watchdog timer interval.

Prototype: `VOID S626_WatchdogPeriodSet( HBD hbd, WORD interval );`

Parameter	Type	Description
hbd	HBD	Board Handle.
interval	WORD	Enumerated watchdog time interval. Takes one of the following values: 0 = 0.125 seconds, 1 = 0.5 seconds, 2 = 1.0 second, 3 = 10 seconds.

Returns: None.

Notes: This function establishes the watchdog timer interval. The watchdog timer interval is defined as the maximum elapsed time between calls to S626_WatchdogReset() that ensures no watchdog timeouts.

When the watchdog timer is enabled, the application must call `S626_WatchdogReset()` at a minimum rate determined by the watchdog interval in order to prevent a watchdog timeout.

Example:

```
/////////////////////////////////////////////////////////////////////////////
// Set the watchdog timer interval on board number 0 to 10 seconds.
/////////////////////////////////////////////////////////////////////////////

#define WD_INTERVAL_SEC_10   3

S626_WatchdogPeriodSet( 0, WD_INTERVAL_SEC_10 );
```

4.8.2 S626_WatchdogPeriodGet()

Function: Returns the programmed watchdog timer interval.

Prototype: `WORD S626_WatchdogPeriodGet( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD containing one of the following enumerated watchdog interval values:

0 = 0.125 seconds,
1 = 0.5 seconds,
2 = 1.0 second,
3 = 10 seconds.

Example:

```
//////////////////////////////////////////////////////////////////////////////////
// Fetch and display the watchdog interval from board number 0.
//////////////////////////////////////////////////////////////////////////////////

printf( "Board 0 watchdog interval = " );

switch ( S626_WatchdogPeriodGet( 0 ) )
{
case 0: printf( "0.125 seconds.\n" );
```

```
case 1: printf( "0.5 seconds.\n"   );
case 2: printf( "1 second.\n"      );
case 3: printf( "10 seconds.\n"     );
}
```

4.8.3 S626_WatchdogEnableSet()

Function: Enables/disables the watchdog timer.

Prototype: `VOID S626_WatchdogEnableSet( HBD hbd, WORD enable );`

Parameter	Type	Description
hbd	HBD	Board Handle.
enable	WORD	Specifies whether watchdog timer is to be enabled or disabled. Set to zero to disable the timer, or to any non-zero value to enable the timer.

Returns: None.

Notes: This function controls the enabling of the watchdog timer. Prior to enabling the watchdog timer, the watchdog time interval should be programmed by calling `S626_WatchdogPeriodSet()`.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// On board number 0, set the watchdog timer interval to 1.0 seconds and
// then enable the watchdog timer.
///////////////////////////////////////////////////////////////////////////////

#define WD_INTERVAL_SEC_1    2

S626_WatchdogPeriodSet( 0, WD_INTERVAL_SEC_1 );    // Set WD interval to 1 sec.
S626_WatchdogEnableSet( 0, true );                 // Enable the WD timer.
```

4.8.4 S626_WatchdogEnableGet()

Function: Returns the state of the watchdog timer enable.

Prototype: `WORD S626_WatchdogEnableGet( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD value indicating whether the watchdog timer is enabled. The returned value is *zero* if the watchdog timer is disabled, or *non-zero* if the timer is enabled.

Example:

```
///////////////////////////////////////////////////////////////////////////////
// Display a message indicating whether the board 0 watchdog timer is enabled.
///////////////////////////////////////////////////////////////////////////////

printf( S626_WatchdogEnableGet() ? "Watchdog enabled\n" : "Watchdog disabled\n" );
```

4.8.5 S626_WatchdogReset()

Function: Resets the watchdog timer.

Prototype: `VOID S626_WatchdogReset( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: None.

Notes: After the watchdog interval has been programmed by `S626_WatchdogPeriodSet()` and the watchdog timer has been enabled by `S626_WatchdogEnableSet()`, this function must be called repeatedly to prevent the watchdog timer from timing out. To guarantee that the watchdog timer will not timeout, the maximum elapsed time between any two calls to `S626_WatchdogReset()` must not exceed the time interval that was last programmed by `S626_WatchdogPeriodSet()`.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// On board number 2, reset the watchdog timer to prevent a watchdog timeout.
/////////////////////////////////////////////////////////////////////////////////

S626_WatchdogReset( 2 );      // "Tag" (reset) the watchdog timer.
```

4.8.6 S626_WatchdogTimeout()

Function: Returns the watchdog timeout status.

Prototype: `WORD S626_WatchdogTimeout( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD indicating whether the watchdog timer has timed out. The returned value is *zero* if no timeout has occurred, or a *non-zero* value if a timeout has transpired.

Notes: This function is useful only if the watchdog timer output has not been routed to the PCI bus hardware reset signal. If the watchdog output is routed to the PCI reset signal, a watchdog timeout will generate a system-wide PCI bus reset, thereby returning the Model 626 watchdog timer to its default, disabled state and clearing the watchdog timeout status indicator.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Display a message indicating whether the board 0 watchdog has timed out.
/////////////////////////////////////////////////////////////////////////////////

printf( S626_WatchdogTimeout() ? "Watchdog timeout\n" : "No watchdog timeout\n" );
```

4.9 Battery Functions

4.9.1 S626_BackupEnableSet()

Function: Enables/disables battery backup of the counter circuits.

Prototype: `VOID S626_BackupEnableSet( HBD hbd, WORD enable );`

Parameter	Type	Description
hbd	HBD	Board Handle.
enable	WORD	Indicates whether battery backup is to be enabled or disabled. Set to *zero* to disable battery backup, or to any *non-zero* value to enable battery backup.

Returns: None.

Notes: An optional, external battery must be connected to the board to supply backup power. Refer to the Model 626 hardware manual for details.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Enable battery backup of the counter circuits on board number 0.
/////////////////////////////////////////////////////////////////////////////////

S626_BackupEnableSet( 0, true );
```

4.9.2 S626_BackupEnableGet()

Function: Returns the state of the enable for battery backup.

Prototype: `WORD S626_BackupEnableGet( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD value indicating whether battery backup is enabled for the counter circuitry. The return value is *zero* if battery backup is disabled, or *non-zero* if battery backup is enabled.

Notes: An optional, external battery must be connected to the board to supply backup power. Refer to the Model 626 hardware manual for details.

Example:

```
/////////////////////////////////////////////////////////////////////////////////
// Display the state of the battery backup enable on board number 0.
/////////////////////////////////////////////////////////////////////////////////

printf( S626_BackupEnableGet( 0 ) ? "Backup enabled.\n" : "Backup disabled.\n" );
```

4.9.3 S626_ChargeEnableSet()

Function: Enables/disables battery charging.

Prototype: `VOID S626_ChargeEnableSet( HBD hbd, WORD enable );`

Parameter	Type	Description
hbd	HBD	Board Handle.
enable	WORD	Indicates whether backup battery charging is to be enabled or disabled. Set to *zero* to disable battery charging, or to any *non-zero* value to enable battery charging.

Returns: None.

Notes: This function enables or disables trickle charging of a battery that supplies power to the counter circuits during a system power failure.

An optional, external battery must be connected to the board. Refer to the Model 626 hardware manual for details.

Example:

```
////////////////////////////////////////////////////////////////////////////////
// Enable charging of the backup battery connected to board number 3.
////////////////////////////////////////////////////////////////////////////////

S626_ChargeEnableSet( 3, true );
```

4.9.4 S626_ChargeEnableGet()

Function: Returns the state of the enable for the battery charger.

Prototype: `WORD S626_ChargeEnableGet( HBD hbd );`

Parameter	Type	Description
hbd	HBD	Board Handle.

Returns: WORD value indicating whether battery charging is enabled for the backup battery. The return value is *zero* if battery charging is disabled, or *non-zero* if battery charging is enabled.

Notes: An optional, external battery must be connected to the board. Refer to the Model 626 hardware manual for details.

Example:

```
////////////////////////////////////////////////////////////////////////////////
// Display a message indicating whether battery charging is enabled on board 3.
////////////////////////////////////////////////////////////////////////////////

printf( S626_ChargeEnableGet( 3 ) ? "Charging enabled.\n" : "Charging disabled.\n" );
```

4.10 Interrupt Functions

4.10.1 S626_InterruptEnable()

Function: Enables/disables the board's master hardware interrupt.

Prototype: `VOID S626_InterruptEnable( HBD hbd, WORD enable );`

Parameter	Type	Description
`hbd`	`HBD`	Board handle.
`enable`	`WORD`	Indicates whether the master hardware interrupt is to be enabled or disabled. Set to *zero* to disable the interrupt, or to any *non-zero* value to enable the interrupt.

Returns: None.

Notes: The Model 626 driver implements hardware interrupt processing by means of a secondary thread, referred to as the *interrupt thread*, which executues concurrently with the main driver thread. The interrupt thread is created when a board is declared to the driver by calling `S626_OpenBoard()` with a non-zero `callback` address. A separate interrupt thread is launched for each board that is declared to the driver.

When launched, the interrupt thread performs various initialization functions and then enters a dormant state. The thread remains in the dormant state while it waits for a hardware interrupt. When an interrupt occurs, the thread awakens, services the interrupt, and then returns to the dormant state to wait for the next interrupt. The interrupt thread is terminated when the board becomes unregistered.

Two services are provided by the interrupt thread in response to a hardware interrupt:

1. The master hardware interrupt enable on the Model 626 board is masked (disabled). This prevents pending interrupt requests from causing nested interrupts while an interrupt service is in progress.
2. An application interrupt service routine (ISR) callback function is invoked so that application-specific interrupt handling can be performed. Note that the ISR callback function executes on the interrupt thread, *not* on the main driver thread.

At a minimum, the application's ISR callback function must determine which onboard resources caused the interrupt, negate the pending interrupt requests for those resources, and then unmask the master board interrupt by calling `S626_InterruptEnable()`. In most applications, a separate callback function is provided for each board in order to quickly resolve the resources that need servicing.

The master board interrupt is masked in response to PCI bus resets or execution of the `S626_OpenBoard()` function. Applications that employ Model 626 interrupts must call `S626_InterruptEnable()` to unmask the master interrupt after either of these events has occurred. Also, because the master interrupt is automatically disabled by the interrupt thread, `S626_InterruptEnable()` is usually invoked by the application at the end of the ISR callback function.

`S626_InterruptEnable()` may be called to temporarily disable interrupts during a *critical segment*, which is defined here as any program sequence that must not be interrupted. This is often required in applications that share memory or other hardware resources among multiple threads.

Example: See the examples in Section 4.11.

4.10.2 S626_InterruptStatus()

Function: Returns the interrupt status of all counter and DIO (digital I/O) channels.

Prototype: `VOID S626_InterruptStatus( HBD hbd, WORD *status );`

Parameter	Type	Description
`hbd`	`HBD`	Board Handle.
`status`	`WORD*`	Address of an array of WORDs that will receive the interrupt status.

The `status` argument is the address of an array of four WORD values that will be populated with interrupt status information; this array will receive interrupt status for all interrupt sources on the specified board.

Table 10 shows the organization of the status information that is loaded into the target WORD array by this function. The first three WORD elements of the array receive the interrupt status of the DIO channels, while the fourth element receives the interrupt status of the counter channels.

Table 10: Organization of WORD array populated by S626_InterruptStatus().

Array Index	Bit Position															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	DIO 15	DIO 14	DIO 13	DIO 12	DIO 11	DIO 10	DIO 9	DIO 8	DIO 7	DIO 6	DIO 5	DIO 4	DIO 3	DIO 2	DIO 1	DIO 0
1	DIO 31	DIO 30	DIO 29	DIO 28	DIO 27	DIO 26	DIO 25	DIO 24	DIO 23	DIO 22	DIO 21	DIO 20	DIO 19	DIO 18	DIO 17	DIO 16
2	0	0	0	0	0	0	0	0	DIO 39	DIO 38	DIO 37	DIO 36	DIO 35	DIO 34	DIO 33	DIO 32
3	OVR 2B	OVR 2A	OVR 1B	OVR 1A	OVR 0B	OVR 0A	IDX 2B	IDX 2A	IDX 1B	IDX 1A	IDX 0B	IDX 0A	0	0	0	0

The following abbreviations are used in Table 10:

Abbreviation:	is the interrupt request status for:
DIO xx	DIO channel xx captured an edge.
OVR xx	Counter xx captured an overflow.
IDX xx	Counter xx captured an index.

A logic one in any bit position indicates that the associated resource is requesting interrupt service. For example, a logic one in bit number 12 of the array's fourth WORD element indicates that counter 1A has overflowed and is requesting interrupt service.

Returns: None.

Notes: This function is typically used by application interrupt callback functions to quickly determine which onboard resources are requesting interrupt service.

Example: See the examples in Section 4.11.

4.11 Interrupt Programming Examples

4.11.1 Example: DIO Interrupts

This example is a simple application that demonstrates how to use DIO interrupts.

This sample application consists of two functions that execute on two separate threads: a primary application function and an interrupt handler. The interrupt handler counts interrupts occuring on DIO channels 0 through 15. Once per second, the main function displays the accumulated interrupt counts and resets the counts to zero.

```
//////////////////////////////////////////////////////////////////////
// Count the interrupts occurring on board 0, DIO channels 0-15.
//////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>
#include "Win626.h"

VOID AppISR();       // Function prototype.

// Array of zeros for fast resetting of interrupt counters.
const DWORD zeros[] = { 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0 };

// Interrupt counters.
DWORD IntCounts[16] = { 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0 };

// Synchronization structure for thread-safe execution.
CRITICAL_SECTION  CriticalSection;

//////////////////////////////////////////////////////////////////////
//  APPLICATION MAIN FUNCTION.

VOID main()
{
    DWORD counts[16];    // Cached copy of interrupt counts.

    // Initialize the thread synchronization structure.
    InitializeCriticalSection( &CriticalSection );

    // Link to S626.DLL.
    S626_DLLOpen();

    // Declare board to driver and launch interrupt thread.
    S626_OpenBoard( 0, 0, AppISR, THREAD_PRIORITY_ABOVE_NORMAL );
    if ( S626_GetErrors( 0 ) )
    {
        printf( "ERROR: problem opening board.\n" );
        S626_DLLClose();    // Unlink from S626.DLL.
        return;
    }

    // Reset all DIO interrupt counters to zero.
    memcpy( IntCounts, zeros, sizeof( zeros ) );

    // Enable event capturing and interrupts on DIO channels 0-15.
    S626_DIOCapEnableSet( 0, 0, 0xFFFF, true );
    S626_DIOIntEnableSet( 0, 0, 0xFFFF );

    // Enable board master interrupt.
    S626_InterruptEnable( 0, true );

    // Repeat for 10 seconds ...
```

```
    for ( int seconds = 0; seconds < 10; seconds++ )
    {
        // Suspend main thread for one second.
        Sleep( 1000 );

        // Copy, and then reset the interrupt counts.  This is a critical section because
        // interrupt counters are shared by both the main thread and the interrupt thread.
        EnterCriticalSection( &CriticalSection );       // * Start thread-safe section --------
        memcpy( counts, IntCounts, sizeof( counts ) );  // * Cache a copy of counters.
        memcpy( IntCounts, zeros, sizeof( zeros ) );    // * Reset counters to zero.
        LeaveCriticalSection( &CriticalSection );       // * End thread-safe section ----------

        // Display cached interrupt counts.
        printf( "\f**** INTERRUPT COUNTS ****\n" );
        for ( int i = 0; i < 16; i++ )
            printf( "DIO %2d counts = %8d", i, counts[i] );
    }

    // Unlink from S626.DLL.
    S626_DLLClose();

    // Delete the critical section.
    DeleteCriticalSection( &CriticalSection );
}

//////////////////////////////////////////////////////////////////////
//  ISR CALLBACK FUNCTION.

VOID AppISR()
{
    WORD IntStatus[4];     // Array that receives interrupt status.

    // Cache a copy of DIO channel 0-15 interrupt request (IRQ) status.
    S626_InterruptStatus( 0, IntStatus );       // Fetch IRQ status for all sources.
    register WORD status = IntStatus[0];        // Cache DIO 0-15 IRQ status.

    // Tally DIO 0-15 interrupts.
    register DWORD *pCounts = IntCounts;        // Init pointer to interrupt counter.
    EnterCriticalSection( &CriticalSection );   // *  Start thread-safe section -----------
    for ( register WORD mask = 1; mask != 0; pCounts++ )
    {                                           // *
        if ( status & mask )                    // *  If DIO is requesting service ...
            ( *pCounts )++;                     // *    increment DIO's interrupt counter.
        mask += mask;                           // *  Bump mask.
    }                                           // *
    LeaveCriticalSection( &CriticalSection );   // *  End thread-safe section -------------

    // Negate all processed DIO interrupt requests.
    S626_DIOCapReset( 0, 0, status );

    // Unmask board's master interrupt enable.
    S626_InterruptEnable( 0, true );
}
```

4.11.2 Example: Counter Interrupts

This example is a simple application that demonstrates how to use counter interrupts.

The application consists of two functions that execute on separate threads: a primary application function and an interrupt handler. The main function initializes the frequency counter, consisting of counter pair 0A/0B, and then sleeps for ten seconds. While the main thread sleeps, the interrupt handler fetches and displays acquired frequency data once per second.

```c
/////////////////////////////////////////////////////////////////////
//  Sample application: interrupt-driven frequency counter.
/////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>
#include "Win626.h"

#define COUNTER   CNTR_0A    // Counter A (and implicitly, B) channel to use.

// Function prototypes.
VOID CreateFreqCounter( HBD hbd, WORD CounterA, WORD GateTime );
VOID AppISR();

/////////////////////////////////////////////////////////////////////
// APPLICATION MAIN FUNCTION.

VOID main()
{
    // Link to S626.DLL.
    S626_DLLOpen();

    // Declare Model 626 board to driver and launch the interrupt thread.
    S626_OpenBoard( 0, 0, AppISR, THREAD_PRIORITY_ABOVE_NORMAL );
    if ( S626_GetErrors( 0 ) )
        printf( "ERROR: problem opening board.\n" );
    else
    {
        // Configure counter A/B pair as a frequency counter, gate time = 1 second.
        // Note: the CreateFreqCounter() function is shown in Section 4.7.6 on page 42.
        CreateFreqCounter( 0, COUNTER, 1000 );

        // Enable interrupts in response to captured counter A overflows.
        S626_CounterIntSourceSet( 0, COUNTER, INTSRC_OVER );

        // Enable board's master interrupt.
        S626_InterruptEnable( 0, true );

        // Sleep for ten seconds.
        Sleep( 10000 );
    }

    // Disconnect from S626.DLL.
    S626_DLLClose();
}

////////////////////////////////////////////////////////////////////////////
// ISR CALLBACK FUNCTION.  Since counter 0A is the only enabled interrupt
// source, there is no need to check for other interrupt requestors.

VOID AppISR()
{
    // Seconds counter.
    static WORD secs = 0;

    // Clear counter A overflow capture flag to negate the interrupt request.
    S626_CounterCapFlagsReset( 0, COUNTER );

    // Read and display measured frequency from counter B.  Note that the first
    // acquisition is not valid and should be ignored.
    printf( "%2d: Frequency (Hz) = %8d.\r", ++secs, S626_CounterReadLatch( 0, COUNTER + 3 ) );
```

```
    // Enable board's master interrupt.
    S626_InterruptEnable( 0, true );
}
```